Workshop on

Hardware Support for Objects

And

Microarchitectures for Java

In conjunction with ICCD'99

Austin, Texas
October 10, 1999

MESSAGE FROM WORKSHOP CO-CHAIRS

Most modern programming languages and techniques include object-oriented methods. However, mainstream computer architectures have not acknowledged the presence of objects. With the widespread use of object-oriented programming languages and techniques, it is becoming important for computer architects to acknowledge the existence of these methods and their impacts on execution (including high object allocation rates, the impact of garbage collection, dynamic binding of calls to methods, and dynamic assembly of programs at run time from components obtained from disparate sources).

Java is an exciting new object-oriented technology. Hardware for supporting objects and other features of Java such as multithreading, dynamic linking and loading is the focus of this workshop. The impact of Java's features on micro-architectural resources and issues in the design of Java-specific architectures are interesting topics that require immediate attention of the research community.

The purpose of this workshop is to draw together researchers and practitioners concerned with hardware support for objects and Java implementations for a stimulating exchange of views. To the organizers' best knowledge, this is the first event of its kind, and as such is an attempt to begin the task of building a community in this field. We thank all the program committee members, the authors and the invited panelists for helping us start this process. Also, we would like to thank the ICCD organizers and Prof. Craig Chase, in particular, for their support to this workshop. We hope you will enjoy this workshop as much as we did in organizing this.

Mario Wolczko
Sun Microsystems

Vijaykrishnan Narayanan
Pennsylvania State University

Organizers

Workshop Co-Chairs

Vijaykrishnan Narayanan, Pennsylvania State Univ.
Mario Wolczko, Sun Microsystems, Inc.

Program Committee

Timothy Heil, Univ. of Wisconsin, Madison
Lizy John, Univ. of Texas at Austin
Vijaykrishnan Narayanan, Pennsylvania State Univ.
Nagarajan Ranganathan, Univ. of South Florida
Mario Wolczko, Sun Microsystems, Inc.

TABLE OF CONTENTS

2:15-3:45p.m. Session 4. ***Microarchitectures for Java***

Chair: Vijaykrishnan Narayanan, Pennsylvania State University

- *VLSI Architecture Using Lightweight Threads (VAULT),*
Ian Watson, Greg Wright, Ahmed El-Mahdy

- *A two step approach in the development of a Java Silicon Machine (JSM) for small embedded systems,*
Hagen Ploog, Ralf Kraudelt, Nicco Bannow, Tino Rachui, Frank Golatowski, Dirk Timmermann

- *Quantitative Analysis for Java Microprocessor Architectural Requirements: Instruction Set Design,*
M. Watheq EL-Kharashi, Fayez ElGuibaly, Kin F. Li

4:00-5:30p.m. Panel Session: ***Java Virtual Machines: What can hardware offer to support them?***

Panelists:
David Hardin, Ajile Systems, Inc.
Jim Smith, University of Wisconsin, Madison
Marc Tremblay, Sun Microsystems, Inc.
Moderator: Mario Wolczko, Sun Microsystems, Inc.

Session 1

Innovations in Memory System Design

A Case for Using Active Memory to Support Garbage Collection

Sylvia Dieckmann and Urs Hölzle
University of California, Santa Barbara
{sylvie,urs}@cs.ucsb.edu

Abstract

Most modern programming languages require efficient automatic memory management (garbage collection, GC) as part of the runtime system. Since GC is very memory intensive it can potentially suffer significantly from poor memory access times. Unfortunately, memory performance improves at a slower pace than processor speed, making memory accesses relatively more expensive in the future. Active Memory architectures aim to overcome this problem by placing additional computational power in memory, thus allowing the application to execute small but memory-intensive functions closer to the data and in parallel. The goal is to improve latency and bandwidth for programs that can otherwise suffer from slow memory accesses.

To date, Active Memory has been studied only with databases, image processing, arithmetic computations, and other very regular applications. In this paper, we propose to analyze its impact on garbage collection. We are convinced that garbage collection too will profit from this architecture since GC is simple, repetitive, easy to partition into offloadable functions, and its performance depends crucially on fast memory access. We describe a possible incarnation of an Active Memory architecture suitable for GC support and argue why GC should benefit from such an architecture.

1. Motivation

Efficient and reliable garbage collection (GC) is an essential part of most modern (especially object-oriented) programming languages. GC relieves the programmer from error prone explicit deallocation, thus preventing memory leaks or early deallocation. But GC performance greatly depends on fast memory access, which can pose a challenge not only to the GC implementation but also to the design of the underlying machine. For example, a simple mark&sweep collector first identifies and marks all live objects and than reclaims (sweeps) the unmarked space. Both phases are very memory intensive since they require touching the entire heap (or at least all live objects) and thus can potentially suffer significantly from poor memory performance. Traditional techniques designed to improve memory performance do not always work for GC. For example, GC can have a negative effect on the cache hit rate because it evicts all application-related entries from the cache. Even worse, GC itself has a poor hit rate because each collection dereferences all pointers at most once. Techniques such as prefetching that exploit access patterns often fail because memory access during GC follows pointer chains and can be very irregular. [10, pp. 284]

Alas, memory performance, namely access latency and bandwidth, is an issue of great concern for modern machines. Already, the cost of processor-memory communication has a significant impact on overall performance. Most researchers agree that it will become even more important in the future, since (1) processor speed is growing at a faster pace than memory performance, and (2) the I/O connections used to deliver this data to the processors have limited bandwidth [15, 28, 53, 71]. For example, Hennessy and Patterson estimate that since 1985 CPU performance has grown by 50% every year whereas DRAM access times have improved by only around 7% per year [19]. With faster processors but similar memory latencies and bandwidths, it becomes harder to feed the processing unit with useful input data to run at peak speed and memory accesses become relatively more expensive. (In the literature this problem is typically referred to as the increasing Memory-Processor Performance Gap.) In addition, modern applications are putting stronger demands on the memory system as data sets grow larger [1, 36] and object-oriented and pointer-based programs with irregular access patterns and automatic memory control are becoming more and more important

We can address the problem of increasing memory access costs in two ways. *Processor-centric* optimizations like prefetching, speculation, multilevel caches and out-of-order execution aim to better cope with the existing bandwidth and latency, for example, by exploiting locality in access patterns. They tend to make things worse, however, if the executed application does not express the expected regularities, which is often the case with modern object-oriented and pointer-based programs. Especially during GC, memory access follows pointer chains and can be very irregular. The cache hit rate can actually deteriorate in the presence of GC because every iteration evicts all application related entries from the cache. Even worse, GC itself has a poor hit rate since most algorithms dereference each pointer only once during each GC phase, thus defeating the advantage of caching data.

In contrast, *memory-centric* approaches hand attempt to move at least part of the computation closer to the data it processes, thus actively improving latency and bandwidth. This concept is represented by a new breed of hardware, *Active Memory* [6, 16, 17, 23] (or the closely related *Active Disks* [1, 22]) which aims to enhance the memory unit with computational logic so that it can take over some or all of the processor's work. Often, Active Memory is implemented by integrating additional logic into the DRAM chip, thus allowing the main processor to offload small functional units called *memlets*[1] to be computed directly in memory. For example, the Active Pages model proposed by Oskin et al. [16] suggests assigning a small embedded processor or an array of FPGAs to every 512 Kbytes of DRAM.

We believe that GC is very likely to benefit from Active Memory. Most GC algorithms are composed of highly repetitive and simple components which can easily be offloaded to an in-memory processor. This is especially true for non-incremental algorithm that stop the actual application (*mutator*) during the duration of the collection. By delegating GC to Active Memory, one could not only reduce the amount of data passed between processor and memory but also parallelize the inner loop of the collector.

We therefore suggest to study the suitability of Active Memory to support automatic memory management (i.e., garbage collection or simply GC). The ultimate goal of this study is to show that garbage collection—which we believe is crucial for runtime performance and thus deserves special effort—will benefit significantly from the existence of an Active Memory approach in the spirit of Active Pages.

We are currently working on a hardware simulator that would allow us to empirically evaluate the impact of ARAM (our Active Memory model) on the performance of garbage collection in a high-performance Java Virtual Machine. With the help of this simulator, we hope to demonstrate that and how garbage collection can profit from the presence of Active Memory. Moreover, we plan to design a suite of GC algorithms partitioned for ARAM and to analyze the impact of software decisions related to partitioning, page size, allocation strategy, etc.

To the best of our knowledge, none of the groups working on Active Memory has made an attempt yet to utilize this architecture for garbage collection or even for language support in general. The idea of Active Memory is still relatively new and few proposals have actually been implemented to date. Those that were evaluated—either by simulation or with a prototype—were usually tested with relatively regular applications from the areas of databases, image processing and arithmetic computation only. Although these studies generally show promising speedups for the selected type of applications as well as technical feasibility of Active Memory hardware, further tests with more sophisticated programs are required to show general applicability.

[1] None of the current proposals actually uses the term *memlet*. However, Acharya et al. [1] refer to *disklets* as code that is offloaded and executed on disk. Accordingly, we refer code that is executed in memory as memlets.

2. Active Memory

Since the research community became interested in Active Memory architectures a few years ago, several models have been proposed. Most approaches differ significantly in terms of expected benefits, targeted applications, design complexity, software effort, and technical feasibility. Unfortunately, a thorough discussion of some of these proposals would exceed the scope of this paper. Instead, in this section we sketch an independent model that represents our understanding of Active Memory and serves as a baseline for our project.

Although this model—which we name ARAM—is derived from Active Pages, a model suggested by Oskin et al. [16], it attempts to be more general than that. Despite the ongoing work of several research teams, many aspects of active memory design are not fully understood yet. Part of our planned work is to investigate various hardware modifications and their impact onto GC behavior. Therefore, the ARAM model is meant to eventually cover a large design space. Nevertheless, to provide a first intuitive notion of ARAM and its capabilities we describe a rather concrete incarnation in this section.

2.1 ARAM

The ARAM model described in this section is based on two fundamental principles: (1) traditional applications that do not define memlets must execute—with similar performance—on an ARAM machine (no harm for anybody) and (2) applications that were modified to use memlets must experience a noticeable speedup (better for some). Although the model leaves many issues to be resolved in later design phases, it aims to provide enough detail so that any ARAM implementation will follow these two guidelines.

As in most current computer architectures, our model consists of a main processor (sometimes also called host processor), a set of ARAM chips that are physically separated from the main processor, and a memory bus to connect both units. The model includes a (multilevel) memory hierarchy with caches and virtual address spaces to allow fast access for applications with good locality. At

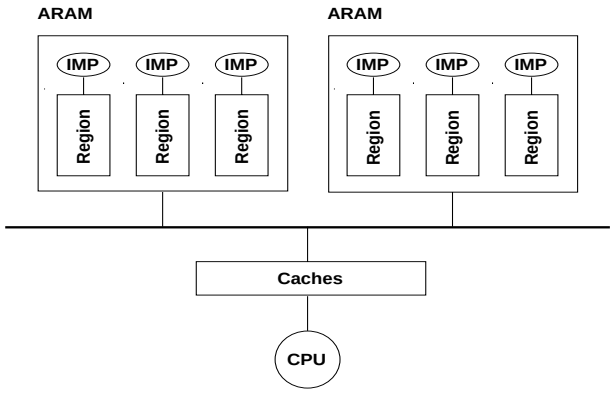


Figure 1. ARAM Architecture

the first sight, an ARAM chip resembles conventional DRAM; but unlike in DRAM, the available storage space in ARAM is divided into one or more units (regions)[1] of equal size. Oskin et al. [16] found 512 Kbyte regions to be most practical for the available hardware. A small embedded RISC processor called *In-Memory Processor* (IMP) is assigned to each unit. To obtain optimal latency and bandwidth, the physical chip layout determines the association between regions and their associated processors. Most likely, the user will be unable to dynamically modify either region size or region-processor association.

We assume that the main processor can access an ARAM cell almost as fast as a cell in a comparable DRAM chip. (Synchronization between main processor and IMP might add a small overhead, though.) As in DRAM, the costs for accesses by the main processor can vary within a certain range depending on the accessed location. But for accesses initiated by memlet code on a certain IMP, the situation is more complex as it most likely depends much more on the accessed memory location relative to the IMP: here we assume that an IMP can access data residing in its own region (called *intra-region access* in the remainder of this document) with significantly shorter latency and larger bandwidth than the main processor. In fact, one of the motivations for transforming memory intensive functions into memlets is to replace main processor accesses with cheaper intra-region IMP accesses. However, accesses to locations that currently belong to the region of another IMP on the same chip (*inter-region, intra-chip accesses*) are likely to be more expensive, although probably still competitive with a conventional main processor access. Any access issued by an IMP and directed to a location on another physical chip (*inter-chip accesses)* will be penalized most. We expect this type of access to be significantly more expensive than a direct access from the main processor.

Note that implementation details as well as actual and relative access costs remain open at this point. Most of these aspects depend on technical conditions and cannot yet be determined anyway. However, any actual ARAM implementation should be guided by two principles: (1) applications that do not use memlets should not suffer from the new architecture and (2) those that do, should experience noticeable speedups due to off-loading of memlets. Therefore, no matter how the final cost model will look, the main processor must be able to access ARAM almost as efficiently as DRAM and IMPs must access at least data in their own domain significantly more efficiently.

Finally, a modified operating system (OS) must provide the traditional OS functionality in combination with ARAM support. For example, it will be in the responsibility of the OS to set up, invoke and manage the memlets (section 2.2 sketches an example API), to synchronize IMPs and main processor, to deliver messages between the processors, to maintain consistency between cache and ARAM, and to maintain a virtual memory system on top of ARAM. The last point is necessary because unlike traditional memory systems, which use only physical addresses, memlet code contains virtual addresses. Consequently, somebody—either the main processor or the IMPs themselves—must be able to resolve these virtual addresses within the memory system.

2.2 Programming Model

The user interface of ARAM provides the standard virtual memory interface extended by a set of functions to allocate regions, define memlets, bind them to a region or a group of regions and activate them. It is in the responsibility of the user to partition an application, i.e. write memlets and invoke them from the code. We use a functional model for this proposal since it seems to be the easiest to integrate into an existing system.

Note that the purpose of this programming model is to help understand the requirements of memlets and application code and to design GC algorithms independent from actual decisions about the underlying hardware. It is meant as an abstraction that hides away hardware details such as region-IMP association and must be general enough to express all GC needs. However, by no means does it determine an actual ARAM implementation.

The user defines memlets as special functions together with the actual application. A memlet operates on a domain given as function parameters during invocation.

[1] We use the term *region* rather than *(active) page* to describe the memory unit directly associated with a single IMP to emphasize the difference between a region/active page and an OS page (i.e. virtual memory page). Active memory is not meant to replace the virtual memory system; usually, regions cover several OS pages.

For now, domains always correspond to one ARAM region; rather than providing a domain parameter with every memlet invocation, the user can bind it to a certain IMP up front. A memlet is invoked by a special instruction such as a store to a memory-mapped device. It can receive as many arguments as it needs. For example, the stored address could point to a memlet header with function pointer, arguments, and IMP identifier. Whenever an IMP detects a write to the magic location, it retrieves this array, determines function and arguments and invokes the memlet. On termination, the memlet sends a signal back to the main processor.

When executing memlet code, the IMP hardware resolves addresses and communicates with other IMPs on the same chip using some protocol. The IMP also provides instructions to indicate the physical location of an address (to determine the access costs). Any memlet can access the entire virtual address space, although accesses to remote locations might be disproportionately expensive. While intra-chip accesses may be resolved in hardware, off-chip accesses might involve software protocols and use the main CPU to relay data to another ARAM chip. While slower than hardware, a software solution would considerably reduce the complexity of ARAM-based systems by eliminating the need for inter-ARAM bus logic.

3. Why GC is Likely to Profit From Active Memory

All garbage collectors perform the same basic task: they determine the set of reachable (i.e., live) objects and reclaim the storage used by all unreachable (dead) objects. Most GC algorithms (with the exception of reference counting) do this by periodically analyzing a snapshot of the heap to detect and reclaim objects that are not longer reachable. Consequently, the collector needs to access all (live) objects, but on each object performs very little computation before it starts visiting the children.[1] This makes GC inherently memory-intensive. In addition, accessing objects by following a pointer chain leads to a very irregular access pattern where each object is visited only once in each phase. Therefore, caches often perform poorly for GC.

Preliminary results from a small study of the effect of garbage collection on cache performance of a Java VM indicate that garbage collection related activity has a significantly higher L1 miss rate than the actual application code (8-16% for GC vs. 6-9% for the application). They also show that the JDK1.1.5 spends up to 30% of its time in garbage collection, which makes good GC performance all the more important. Write misses, although generally not as critical as read misses, rise to about 28% for one application (javac).[2]

To benefit from Active Memory, an application must be partitionable so that enough computational work involving memory accesses can be offloaded and parallelized. We are convinced that GC algorithms generally fulfill this requirement since most memory activity occurs in a tight inner loop. In terms of computational complexity, this loop is simple enough to be offloaded to an IMP with limited power. Although several algorithms require scratch space, one can usually define an upper bound. The inner loop is also likely to benefit from parallelization; for example, Endo et al. [7] studied a parallel mark-sweep collector and reported a significant speedup for parallel marking with work stealing in a shared heap.

Parallelization can become a problem for ARAM if the application contains a high number of inter-region references. In the worst case, every single step could require inter-region communication (e.g., if a chain of pointers is spread over several regions). However, we believe that his risk can be reduced by (1) dividing the heap over regions in accordance to the access order imposed by the collector scheme (e.g., generational GC, Train Algorithm) or (2) by instrumenting a copying collector to rearrange objects in order to reduce region-crossing references.

Finally—at least at this point—partitioning an application to use ARAM is awkward and requires some internal knowledge. However, GC is part of the runtime system, written by a language implementor. Unlike the end user, these system experts can justify spending a great deal of time with low-level optimizations as it will pay off multiple times later during runtime.

To summarize our arguments, we believe that good GC performance is crucial for state-of-the-art OO systems and that memory latency and bandwidth is a significant factor in GC overhead. The overall structure of most GC algorithm is simple, highly repetitive, and memory intensive. Therefore, most algorithms can naturally be divided into memlets executed in ARAM, which would parallelize the collection, improve latency and bandwidth for offloaded memory accesses, and greatly reduce the amount of data

[1] In this respect GC resembles a pointer chase problem.

[2] In most of these experiments we ran the optimized JDK1.1.5 executing memory intensive programs from the SPECjvm98 benchmark suite on a 147 MHz UltraSPARC-I with 16 Kbytes L1 and 512 Kbytes L2 caches. The UltraSPARC family provides hardware registers to count some basic events during execution at no additional costs. By polling these counters before and after each GC one can observe cache access and miss rates of a life application with virtually no impact on the application's performance.
Since polling hardware counters requires modifying source code, we have not yet repeated the same experiments for a more competitive JVM.

transferred to the main processor. This is especially important for GC performance since conventional methods to improve memory performance such as caches do not always suffice for this type of algorithms.

4. References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming model, algorithms and evaluation. In *Proceedings of ASPLOS VIII,* San Jose, CA, October 1998.

[2] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A compiler-managed memory system for RAW machines. In *Proceedings* of ISCA-26, Atlanta, GA, June 1999.

[3] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang. Evaluation of existing architectures in IRAM systems. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember,* Denver, CO, June 1997.

[4] D. Burger, J. Goodman, and A. Kagi. Quantifying memory bandwidth limitations in future processors. In *Proceedings of ISCA-23,* Philadelphia, PA, May 1996.

[5] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of ASPLOS VIII,* San Jose, CA, October 1998.

[6] J. Carter et al. Impulse: Building a smarter memory controller. In *Proceedings of HPCA-5,* Orlando, FL, January 1999. IEEE Computer Society.

[7] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of SC97,* November 1997.

[8] M. Gonçalves. *Cache Performance of Programs with Intensive Heap Allocation and Generational Garbage Collection.* Ph.D. thesis, Princeton University, May 1995.

[9] M. Gonçalves and A. Appel. Cache performance of fast-allocating programs. In *Record of FPCA'95,* June 1995.

[10] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[11] T. Kamada, S. Matsuoka, and A. Yonezawa. Efficient parallel global garbage collection on massively parallel computers. In *Proceedings of SC'94,* pages 79-88, 1994.

[12] K. Keeton, D. Patterson, and J. Hellerstein. A case for Intelligent Disks (IDISKs). In *SIGMOD Record,* 27(3), August 1998.

[13] C. Kozyrakis and D. Patterson. A new direction for computer architecture research. *IEEE Computer,* 31(11):24-32, November 1998.

[14] C. Kozyrakis at al. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer,* 30(9):75-78, September 1997.

[15] S. Nettles and J. O'Toole. Real-time replication garbage collection. In *Proceedings of PLDI'93,* volume 28(6) of ACM SIGPLAN Notices, Albuquerque, NM, June 1993.

[16] M. Oskin, F. Chong, and T. Sherwood. Active Pages: A computation model for intelligent memory. In *Proceedings ISCA'98,* Barcelona, Spain, June 1998.

[17] D. Patterson et al. A case for intelligent RAM: IRAM. *IEEE Micro,* 17(2):34-44, March-April 1997.

[18] D. Patterson et al. Intelligent RAM (IRAM): The industrial setting, applications, and architectures. In *Proceedings of ICCD'97,* TX, October 1997.

[19] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufman, 1994.

[20] M. Reinhold. Cache performance of garbage-collected programs. In *Proceedings of PLDI'93,* volume 28(6) of ACM SIGPLAN Notices, Albuquerque, NM, June 1993.

[21] K. Taura and A. Yonezawa. An efficient garbage collection strategy for parallel programming languages on large scale distributed-memory machines. In *Proceedings of PPoPP-6,* ACM SIGPLAN Notices, pp. 264-275, Las Vegas, NE, June 1997.

[22] M. Uysal, A. Acharya, and J. Saltz. An evaluation of architectural alternatives for rapidly growing datasets: Active disks, clusters, and SMPs. Technical Report, TRCS98-27, Department of Computer Science, University of California, Santa Barbara, October 1998

[23] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw Machines. *IEEE Computer,* 30(9):86-93, September 1997.

[24] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News,* 23(1), March 1995.

[25] B. Zorn. The effect of garbage collection in cache performance. Technical Report, CU-CS-528-91, Department of Computer Science, Campus Box 430, University of Colorado, Boulder, May, 1991.

Tolerating Latency by Prefetching Java Objects

Brendon Cahoon Kathryn S. McKinley

Department of Computer Science, University of Massachusetts, Amherst, MA 01003, {*cahoon,mckinley*}*@cs.umass.edu* *

Abstract

In recent years, processor speed has become increasingly faster than memory speed. One technique for improving memory performance is data prefetching which is successful in array-based codes but only now are researchers applying to pointer-based codes. In this paper, we evaluate a data prefetching technique, called greedy prefetching, for tolerating latency in Java programs. In greedy prefetching, when a loop or recursive method updates an object o, we prefetch objects to which o refers. We describe inter- and intra-procedural algorithms for computing objects to prefetch and we present preliminary results showing its effectiveness on a few, small Java programs. Prefetching improves performance, but there is significant room for further improvement.

1. Introduction

Modern processor speeds continue to significantly outpace advances in memory speed. Even though modern processors use deep memory hierarchies, the disparity between processor and memory speeds results in an under utilization of resources due to memory bottlenecks.

Software controlled data prefetching is a technique for improving memory performance by tolerating latency in the memory hierarchy. Compilers statically analyze programs and insert prefetch instructions to load data into the cache prior to use. Previous research shows the benefits of software prefetching techniques in array-based scientific programs [4, 14, 2, 13]. Prefetching in array-based codes is simpler than in pointer-based codes. Given an array, the size of each element and a regular access pattern, the compiler can compute the address of any element in the array and schedule prefetches to elements in a loop that will be accessed in future iterations. Array-based codes are also amenable to analyses which allow compilers to restructure loops, using techniques such as loop tiling, to improve spatial and temporal locality. Recent work uses prefetching for programs with dynamically allocated data structures [11, 12, 16]. However, this work only considers C programs.

Compilers cannot use the same approach in pointer-based codes because separate dynamically allocated objects are disjoint and the access patterns are less regular and predictable. Given an object *o*, we know the address of objects that *o* references and cannot prefetch arbitrary objects without following pointer chains.

In this paper, we evaluate one simple prefetching technique, called greedy prefetching, on Java programs. Luk and Mowry introduced and evaluated the greedy prefetching algorithm for recursive data structures in C programs [12]. We investigate the applicability and effectiveness of greedy prefetching for Java programs. Our specific contributions include a new intra-procedural data flow analysis for finding objects to prefetch, the use of an inter-procedural analysis to improve our analysis in the presence of recursion, and a preliminary evaluation on object-oriented programs.

Object-oriented programs pose analysis challenges because they mostly allocate data dynamically, contain frequent method invocations, and often implement loops with recursion. We use Vortex, a compiler containing advanced analyses specifically tailored for object-oriented languages [9]. Our preliminary results indicate that greedy prefetching is effective on a few, small object-oriented programs. Also, class analysis and method inlining enable effective greedy prefetching. We plan to implement more sophisticated prefetching techniques in the future.

2. Related Work

In this section, we give a brief summary of related work for improving memory performance of pointer-based codes. Previous work investigating prefetching on pointer-based codes only uses C programs.

Lipasti et. al., present one of the initial evaluations of prefetching pointer-based codes [11]. The technique, called SPAID, generates prefetch instructions for function arguments prior to calls. Results show cache miss rate improvements on several programs.

Luk and Mowry introduce and evaluate the greedy prefetching algorithm using C versions of the Olden benchmarks [12]. The main contribution of our work is to use data flow algorithms rather than a loop-based approach and we extend the analysis for object-oriented features. Our preliminary results show similar performance results to Luk and Mowry on Java programs. Luk and Mowry also introduce history based prefetching and data linearization, and present limited results on hand optimized examples. Roth and Sohi

*This work is supported by NSF grant EIA-9726401, an NSF Infrastructure grant CDA-9502639, Darpa grant 5-21425, and Compaq. Kathryn S. McKinley is supported by an NSF CAREER Award CCR-9624209. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

```
class SList {
  int   data;
  SList next;
  int sum() {
    prefetch(next);
    if (next != null)
      return data + next.sum();
    return data;
} }
```

```
class DList {
  int   data;
  DList next, prev;
  int sum() {
    prefetch(next);
    // prefetch(prev);
    if (next != null)
      return data + next.sum();
    return data;
} }
```

```
class Tree {
  int   data;
  Tree  left, right;
  int sum() {
    prefetch(left);
    prefetch(right);
    int s = data;
    if (left != null) s += left.sum();
    if (right != null) s += right.sum();
    return s;
} }
```

Figure 1. Prefetch examples for singly linked list, doubly linked list, and binary tree

evaluate a hardware/software prefetching approach for tolerating memory latency in pointer-based codes [16]. The technique uses jump-pointer prefetching which is an extension of Luk and Mowry's history-pointer technique. Roth and Sohi present results using the C version of the Olden benchmark suite. We intend to extend these techniques for Java in the future.

Several researchers improve the memory performance of pointer-based programs by rearranging data at run time [3, 6, 7, 8, 18]. Rubin, Bernstein, and Rodeh combine data reorganization and a different type of greedy prefetching to improve performance on a small C kernel [17].

3. Greedy Prefetching

We extend Luk and Mowry's algorithm for prefetching object-oriented languages such as Java. During the traversal of a linked data structure, greedy prefetching attempts to prefetch objects that will be accessed in the future. Its major limitation is that it can only schedule prefetch instructions for objects directly connected to the current object.

Figure 1 shows simple class definitions for a singly linked list, a doubly linked list, and a binary tree (we use the examples to illustrate greedy prefetching and not as good examples of object-oriented programming). Each class contains a `sum` method which adds the elements in the data structure. In the example, we insert a prefetch instruction for the *next* object in the linked list. We cannot prefetch objects further ahead because we do not know the address of future objects. Prefetching two objects ahead, `prefetch(this.next.next)`, requires the address of `this.next` which is unknown until the program dereferences `this`.

Achieving the full benefits of prefetching requires the computation time between the prefetch and use of the object to be greater than or equal to the memory access time to completely hide the latency. However, even if the computation time is less than the memory access time, the prefetch can partially hide the latency. In the linked list example, we only partially hide the read latency of `next` if the cost of the addition and function call is less than the cost of a memory access. Similarly, we typically only partially hide the latency of the prefetch of `left` in the binary tree example. However, since we also prefetch `right`, we may completely hide its memory cost.

The greedy prefetch algorithm consists of two parts; a phase which finds objects to prefetch followed by a phase which schedules the prefetch instructions. The algorithm is greedy because we do not perform any analysis to determine if an object is already in the cache and we try to prefetch as much as possible. Our algorithm uses both intra-procedural and inter-procedural data flow analysis to find object traversals in loops and recursive calls.

3.1. Detecting Recurrent Object Updates

A recurrent object update is a statement of the form, `o = o.next`, occuring in a loop or recursive method call. In Figure 1, `next.sum()` and `left.sum()` are examples of recurrent object updates occurring in recursive calls.

Detecting recurrent object updates is similar to finding loop induction variables. Previous algorithms for finding induction variables are either loop based [1] or use static single assignment (SSA) form [19]. We present a traditional data flow analysis approach to finding recurrent object updates.

The algorithm for detecting recurrent object updates requires both intra- and inter-procedural analysis. The intra-procedural analysis phase detects recurrent object updates in loops. The inter-procedural analysis detects recurrent object updates in recursive method calls. Luk and Mowry do not perform inter-procedural analysis and only identify self recursive calls.

Our intra-procedural data flow analysis is a forward, iterative traversal that uses a three stage lattice to capture recurrent object updates at each point in the program. We define a function to map each object to a lattice value at each point in the program.

Not recurrent. The top element indicates an object is not recurrent.

Possibly recurrent. The first time we process an object it is potentially recurrent.

Recurrent. The bottom element indicates an object is recurrent.

At each store expression in the program, we define a transfer function which assigns objects to lattice values.

- If the store expression is a field assignment of the form `o = p.next`, when *next* is an object reference and *p* is *not recurrent*, then we mark `o` *possibly recurrent*. If `p` is *possibly recurrent*, then `o` is *recurrent*.

Table 1. Olden Benchmark Suite

Name	Main Data Structure(s)	LOC	Methods	Bytecode Len.	Inputs	Inst. Issued	Total Memory
mst	array of lists	206	39	1452	512 nodes	406M	10.4MB
perimeter	quad tree	219	44	1717	4K x 4K image	239M	4.3MB
treeadd	binary tree	66	11	474	1M nodes	264M	24.3MB
tsp	binary tree, linked list	273	15	1711	60,000 cities	1106M	7.2MB
voronoi	binary tree	612	89	4138	20,000 points	1043M	19.5MB

- If the store expression is an object assignment, `o = p`, then assign the value of `p` to `o`.
- For all other store expressions, `o = expr`, we assign the value *not recurrent* to `o`.

At termination, *objects at each program point* belong to one of the 3 lattice values. The data flow merge function ensures the ordering $not\ recurrent \succ possibly\ recurrent \succ recurrent$.

We use the *possibly recurrent* value to detect looping structures. The first time we analyze a loop, an object, `o`, occurring on the LHS of a field reference becomes *possibly recurrent* (*e.g.*, `o = b.next`). On the second iteration of the analysis, the object becomes *recurrent* if the base object of the field reference (*i.e.*, `b`) is also *possibly recurrent*. If `b` is *not recurrent*, then `o`'s value remains the same. The algorithm incorrectly marks objects in loop invariant expressions as *recurrent* (*e.g.*, `t` in `o=p.next; t=o.next`). Moving loop invariant expressions out of loops eliminates this problem.

We also track the fields used in the recurrent object updates. In Figure 1 for example, we record that `next` is the only field involved in the recurrent object update for the traversal of the doubly linked list.

3.2. Scheduling Prefetch Instructions

We greedily schedule prefetch instructions for objects our algorithm finds *recurrent* during the analysis phase. We insert prefetch instructions at the earliest point when we know the base object is not null. The intra-procedural class analysis in Vortex indicates when objects are not null. In Figure 1, the `this` pointer is the base object and we know it is not null upon entering `sum`.

The scheduling phase uses the field information the analysis phase computes to only schedule prefetches for fields involved in recurrent object updates. For the doubly linked list in Figure 1, we only generate a prefetch of the `next` field and not the `prev` field. Luk and Mowry's algorithm generates both prefetches since they do not track the fields used in the recurrent updates.

During scheduling we perform a simple alias analysis to ensure the scheduler only generates a single prefetch instruction for groups of aliased recurrent objects. Temporary objects cause aliases when used in sequences such as `p=o.next; o=p;`. In a loop, we mark both `o` and `p` as *recurrent*, but we only generate a prefetch for one of the objects.

3.3. Inter-procedural Algorithm

We use an inter-procedural algorithm to find recurrent object updates occurring in recursive method calls. Using an inter-procedural data flow analysis is an extension of Luk and Mowry's original algorithm which is only able to detect self recursive function calls.

The inter-procedural algorithm is a top-down, context-sensitive traversal of the call graph. A context-sensitive algorithm enables the analysis phase to determine the fields used in recurrent object updates. A context-insensitive algorithm cannot track the recurrent fields because distinct method calls are treated similarly. For example, in Figure 1, a context-sensitive analysis determines that `this.left` and `this.right` are both recurrent fields in the recursive method, `sum`. A context-insensitive analysis only analyzes `sum` once and will not determine that both `left` and `right` are recurrent fields.

The inter-procedural analysis uses our intra-procedural analysis to compute the recurrent objects within a procedure. At each call site, we map the recurrent lattice values from each actual to each formal. Then, we analyze the method using the intra-procedural analysis. Recursive calls cause the analysis to iterate until the recurrent status of the formals reaches a fixed point.

3.4. Summary of Extensions for Java

Object-oriented languages contain features which are potentially problematic for the greedy prefetch algorithm. We believe inter-procedural analysis improves the effectiveness of greedy prefetching because object-oriented programs often use recursion to express looping constructs. In Figure 1, the `sum` method uses recursion to sum the objects in the linked list. In C, programmers typically use a while statement for the same function.

To improve the effectiveness of greedy prefetching in Java, we run our algorithm after performing class hierarchy analysis and inlining. One use of class hierarchy analysis enables virtual method invocations to be transformed into direct function calls which improves our inter-procedural analysis and also improves inlining [10]. We rely upon inlining to remove unnecessary method calls that encapsulate references to potential recurrent fields. The following is a typical Java code sequence for traversing a linked list.

```
Enumeration e = list.elements();
while (e.hasMoreElements()) {
  List l = (List)e.nextElement();
```

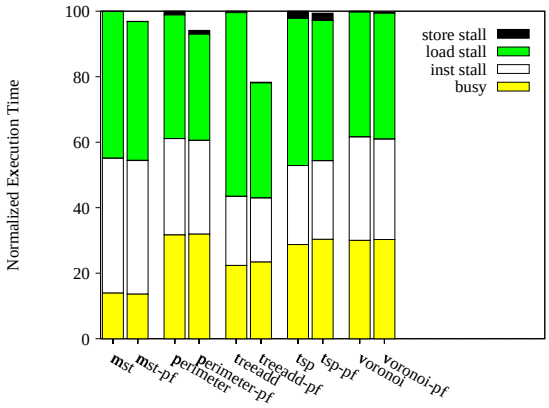


Figure 2. Performance of Greedy Prefetching

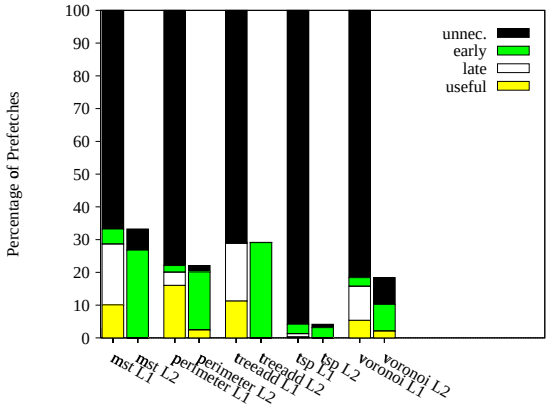


Figure 3. Prefetch Effectiveness

```
  // computation involving l
}
```

If `list` is a linked list with a `next` field, then the expression `e.nextElement` hides the access of `l.next` and the expression `e.hasMoreElements` hides the test for `null`. Inlining eliminates the calls to `hasMoreElements` and `nextElement`.

In the absence of inlining, we can extend the interprocedural data flow analysis to track methods returning fields and use the information to check for recurrent objects. We plan on extending our analysis in the future, but inlining appears to provide most of this benefit.

4. Experimental Results

We implement the greedy prefetching algorithm in the Vortex optimizing compiler [9]. We use Vortex to compile Java programs, perform object-oriented and traditional optimizations, and generate Sparc assembly code.

We present preliminary results using several programs from the Olden benchmark suite [5]. Researchers have used the Olden suite to evaluate optimizations for pointer-based programs [7, 12, 16]. Table 1 lists the Olden benchmarks we use in our experiments along with characteristics about each program. We translated the programs, originally written in C, to Java using an object-oriented style. We compile the programs using JDK 1.1.6. The lines of code (LOC) number excludes comments and blank lines. The bytecode length is the size of the code segments in bytes and not the number of instructions in the programs. We compute the total memory using `totalMemory()` and `freeMemory()` from the `Runtime` class. We disable garbage collection during all our experiments.

We use RSIM to perform a detailed cycle by cycle simulation of our programs [15]. RSIM models a modern out-of-order processor based upon the MIPS R10000. The default processor runs at 300 MHz, issues up to 4 instructions per cycle, and has a 64 entry instruction window. The functional units include 2 ALU, 2 FP, 1 branch, and 2 address units. The instruction window has 64 entries. We use the default values for most of the parameters except for the cache hierarchy. The following table lists the memory hierarchy RSIM parameters we use in our experiments. The default cache sizes are small for modern processors, but match our data sizes and decrease simulation times.

L1 Cache	16 KB, direct WT, split
L2 Cache	64 KB, 4-way, WB, unified
Request Ports	2
Line Size	32 B
L1/L2/Mem hit time	1/12/60 cycles
Cache Miss Handlers (MSHR)	8,8 (L1, L2)

Figure 2 shows preliminary performance results of greedy prefetching. We normalize the results to the execution time, in cycles, of the programs when we do not perform prefetching. We use the RSIM convention to account for busy and stall cycles. We mark a cycle *busy* if the processor retires 4 instructions (the maximum). Otherwise, the first instruction that cannot be retired by the cycle accounts for a *stall*. Figure 2 shows improvements of 3% (mst), 6%, (perimeter), 12% (treeadd), and <1% (tsp, voronoi). Improvements are due to fewer load stalls in the programs. Even after prefetching, the percentage of load stalls remains quite high.

Figure 3 provides insight into the effectiveness of prefetching by dividing the prefetches into various categories. A *useful* prefetch arrives on time and is accessed. The latency of a *late* prefetch is only partially hidden because a cache miss occurs while the memory system retrieves the datum. The cache replaces an *early* prefetch before the use of the datum. An *unnecessary* prefetch hits in the cache or is coalesced into an MSHR. Figure 3 categorizes the prefetches for both L1 and L2 prefetches. We scale the graph for the L2 prefetches to the percentage of requests to the L2 cache. Useful, late, and early prefetches require accesses to the next level in the memory hierarchy. For each program, prefetches to the L1 cache contain a small number of useful and late prefetches. However, many of the prefetches are unnecessary because they hit in the L1 cache. Most prefetches are early in the L2 cache because the cache is small, unified, and write-back so much of the data are replaced.

Table 2. Cache Statistics with and without Prefetching

Program	Reads	L1 Hit	L1 Miss (%)			L2 Hit	L2 Miss (%)			Prefetches		
	(M)	(%)	conf.	cap.	coal.	(%)	conf.	cap.	coal.	static	dyn. L1 (M)	dyn. L2 (M)
mst	13.5	78.3	0.6	13.8	7.3	35.8	7.8	57.3	0			
w/pf	14.2	81.5	0.6	8.9	9.0	43.8	10.5	45.7	0	8	2.235	.743
perimeter	30.4	95.9	0.9	0.9	2.3	53.4	4.5	42.0	0			
w/pf	30.4	96.8	0.6	0.7	1.9	53.0	6.6	40.3	0	8	.32	.071
treeadd	11.5	81.6	0.1	7.1	11.2	3.1	0.9	96.0	0			
w/pf	11.3	85.2	0.1	1.4	13.3	14.0	5.9	80.1	0	2	2.26	.659
tsp	106.3	97.4	0.6	1.0	1.0	50.1	14.4	35.5	0			
w/pf	126.4	96.5	1.0	0.7	1.8	77.2	3.2	19.6	0	31	25.8	1.08
voronoi	113.8	93.6	1.5	2.2	2.7	59.0	12.6	26.4	2.0			
w/pf	113.3	93.8	1.4	2.1	2.7	56.9	14.4	26.2	2.5	18	.816	.150

Table 2 lists several important cache statistics for each program with and without prefetching. We divide the miss statistics into conflict, capacity, and coalesced misses (cold misses are insignificant). A coalesced reference misses in the cache, but hits in a MSHR. The table also displays statistics on the number of static and dynamic prefetches. The static prefetch numbers do not include 23 prefetch instructions the compiler inserts into the Java library code. In general, prefetching improves the hit rates and reduces capacity misses.

5. Conclusion

Traditional compiler algorithms for improving the cache performance are difficult to perform on languages that mostly allocate memory dynamically. Compiler inserted data prefetching is an effective technique for tolerating latency, even in pointer-based programs. In this paper, we evaluate the usefulness of prefetching in Java programs. We present an intra- and inter-procedural algorithm for a simple prefetching algorithm, called greedy prefetching. Our preliminary results show improvements due to prefetching. However, our results indicate that there is room to improve prefetching effectiveness in Java programs because many prefetch instructions hit in the cache. We plan to continue investigating better prefetching algorithms for Java.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] D. Bernstein, D. Cohen, A. Freund, and D. E. Maydan. Compiler techniques for data prefetching on the PowerPC. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, pages 19–26, Limassos, Cyprus, June 1995.

[3] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *ASPLOS-VIII: Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS-IV: Fourth International Conference on Architecturual Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, Apr. 1991.

[5] M. C. Carlisle and A. Rogers. Software caching and computation migration in olden. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbara, CA, July 1995.

[6] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, Atlanta, GA, May 1999.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Atlanta, GA, May 1999.

[8] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *The 1998 International Symposium on Memory Management*, Vancouver, BC, Oct. 1998.

[9] J. Dean, G. DeFouw, D. Grove, V. Litinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*, pages 83–100, San Jose, CA, Oct. 1996.

[10] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95 Conference Proceedings*, Aarhus, Denmark, Aug. 1995.

[11] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microachitecture*, 1995.

[12] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *ASPLOS-VII: Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, MA, Oct. 1996.

[13] N. McIntosh. *Compiler Support for Software Prefetching*. PhD thesis, Rice University, May 1998.

[14] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V: Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–72, Oct. 1992.

[15] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM reference manual (version 1.0). Technical Report Technical Report 9705, Rice University, Dept. of Electrical and Computer Engineering, Aug. 1997.

[16] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[17] S. Rubin, D. Bernstein, and M. Rodeh. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. In *Compiler Construction, 8th International Conference, CC'99*, pages 259–273. Springer, Mar. 1999.

[18] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, Oct. 1998.

[19] M. Wolfe. Beyond induction variables. In *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Francisco, CA, June 1992.

An Introduction to DMMX (Dynamic Memory Management Extension)

J. Morris Chang, Witawas Srisa-an and Chia-Tien Dan Lo

Department of Computer Science
Illinois Institute of Technology
Chicago, IL, 60616-3793, USA
{chang | sriswit | lochiat} @charlie.iit.edu

Abstract

Automatic Dynamic Memory Management (ADMM) allows programmers to be more productive and increases system reliability and functionality. However, the true characteristics of these ADMM algorithms are known to be slow and non-deterministic. It is a well-known fact that object-oriented applications tend to be dynamic memory intensive. Therefore, it is imperative that the programmers must decide whether or not the benefits of ADMM outweigh the shortcomings. In many object-oriented real-time and embedded systems, the programmers agree that the shortcomings are too severe for ADMM to be used in their applications. Therefore, these programmers while using Java or C++ as the development language decide to allocate memory statically instead of dynamically. In this paper, we present the design of an application specific instruction extension called Dynamic Memory Management eXtension (DMMX) that would allow automatic dynamic memory management to be done in the hardware. Our high-performance scheme allows both allocation and garbage collection to be done in a predictable fashion. The allocation is done through the modified buddy system, which allows constant time object creation. The garbage collection algorithm is mark-sweep, where the sweeping phase can be accomplished in constant time. This hardware scheme would greatly improve the speed and predictability of ADMM. Additionally, our proposed scheme is an add-on approach, which allows easy integration into any CPU, hardware implemented Java Virtual Machine (JVM), or Processor in Memory (PIM).

index terms: automatic dynamic memory management, real-time garbage collector, mark-sweep garbage collector, instruction extension, object-oriented programming

1. Introduction

By early 2000s, many industrial observers predict that the VLSI technology would allow fabricators to pack 1 billion transistors into a single chip that can run at Giga-Hertz clock speed. Obviously, the challenge is no longer how to make billion-transistor chips, but instead, what kind of facilities should be incorporated into the design [5]. The current trend in CPU design is to include application specific instruction sets such as MMX and 3D-now as extensions to basic functionalities. The rationales behind such approaches are obvious. First, space and cost limitations are no longer issues. High-density chips can be manufactured cheaply in current semiconductor technology. Second, these application specific instruction sets are included to alleviate performance bottlenecks in the most commonly used applications such as 3-D graphic rendering and multimedia. These rationales closely follow the corollary of Amdahl's law: *Make the common case fast.* Amdahl's Law reminds us that the opportunity for improvement is affected by how much time the event consumes. Thus, making the common case fast will tend to enhance the performance better than optimizing rare cases [7]. Since the biggest merit of hardware is speed, the significant speedup can be gained through hardware implementations of common cases.

As the popularity of object-oriented programming and graphical user interface increases, applications become more and more dynamic memory intensive. It is well-known among experienced programmers that automatic dynamic memory management functions (i.e. allocation and garbage collection) are slow and non-deterministic. Since object-oriented applications prolifically allocate memory in the heap, it is also no coincident that such applications can run up to 20 times slower than the procedural counterparts. A study has also shown that Java applications can spend 20% of the execution time in dealing with dynamic memory management [1]. Unlike stack or queue, heap is not a well-defined data structure. Allocating memory in the heap often requires some form of search routines. In software approaches to heap management, searching is done in sequential fashion (i.e. linked list search). As the number of existing objects grows, the search time would grow linearly longer as well. Studies have shown that applications written in C++ can invoke up to ten times more dynamic memory management calls than comparable C applications [10]. Apparently, dynamic memory management is a common case in object-oriented programming. With Amdahl's corollary in mind, the need of a high-performance dynamic memory manager is obvious.

Deterministic turnaround time is a very desirable trait for real-time applications. Presently, software approaches to automatic dynamic memory management often fail to yield

predictable turnaround time. The most often used software approach in maintaining allocation status is sequential fit or segregated fit. These two approaches utilize linked-list to keep the occupied chunks or free chunks. With linked-list, the turnaround time often relates to the length of the list. As the linked-list becomes longer the sequential search time would grow longer as well [9]. Similarly, the software approaches to garbage collection also yield unpredictable turnaround time. Basically two of the most common approaches for garbage collection are mark-sweep and copying collector. In both instances, the turnaround time is not deterministic.

According to Nilsen and Schmidt, one of the ways to achieve hard real-time performance for garbage collection is through the hardware support [8]. In this paper, we introduce an application specific instruction extension called Dynamic Memory Management eXtension (*DMMX*) that includes *h_malloc*, *mark*, and *sweep* instructions at the user-level. In *h_malloc*, our high-performance allocation scheme allows allocation to be completed in a few instruction cycles. Unlike software approaches, our scheme is fast and deterministic. To perform garbage collection, the *mark* instruction is invoked repeatedly until all the live objects are marked on a bit-map. Once the marking phase is completed, the *sweep* instruction is called. Since we have a dedicated hardware to perform the sweeping, this phase can be completed in a few instruction cycles.

The remainder of this paper is organized as follow. Section 2 provides a top-level architecture of our instruction set. Section 3 describes the internal structure of the Dynamic Memory Management Unit (DMMU). Section 4 addresses the architectural support issues for the DMMU. Section 5 concludes this paper.

2. Overview of the DMMX

In our proposed Dynamic Memory Management eXtenstion (*DMMX*), there are three user-level instructions, *h_malloc*, *mark*, and *sweep*. These three instructions are used as the communication channels between the CPU and the Dynamic Memory Management Unit (*DMMU*). This DMMU can either be packaged inside CPUs or outside. This unit can also be included inside the hardware implemented Java Virtual Machines (i.e. PicoJava II from Sun Microsystems). The main purpose of the *DMMU* is to take responsibility for managing heap space for all processes in the hardware domain. The proposed *DMMU* utilizes the modified buddy system combined with the bit-map approach to perform constant-time allocation [4]. Usually, each process has a heap associated with it. In the proposed scheme, each heap requires three bit-maps, one for allocation status (*A bit-map*), one for object size (*S bit-map*), and one for marking during the garbage collection (*X bit-map*). It is necessary to place these three bit-maps together all the time, since searching and modification to these three bit-maps are required for each garbage collection cycle. Figure 1 demonstrates the top-level integration of the *DMMU* into a computer system.

Figure 1. The top-level description of a *DMMU*

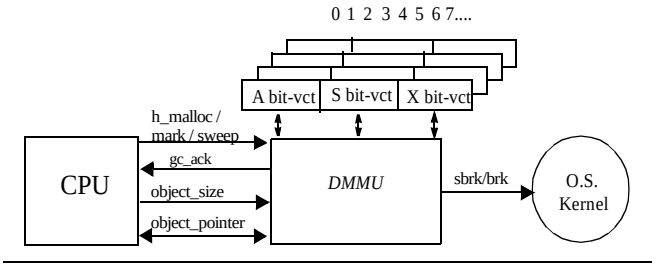


Figure 1 illustrates the basic functionality of the *DMMU*. First, the *DMMU* provides services to CPU by maintaining the memory allocation status inside the heap region of the running process. Thus, the *DMMU* must be able to access the *A bit-map*, *S bit-map*, and *X bit-map* of the running process. Similar to *TLB*, the *DMMU* is shared among all processes. The parameters that the CPU can pass to the *DMMU* are the *h_malloc, mark, or sweep* signal, the *object_size* (for the allocation request), and the *object_pointer*. The operations of the *DMMU* are very similar to the function calls (i.e. *malloc()*) in C language. Thus, *object_pointer* is either returned from the *DMMU* in allocation or passed on to the *DMMU* during the garbage collection process. The *gc_ack* is also returned at the completion of garbage collection cycle. If the allocation should failed, the *DMMU* would make a request to the operating system for additional memory using system call *sbrk()* or *brk()*.

Since the algorithms used in the *DMMU* are implemented through pure combinational logic, the time to perform a memory request or memory sweeping is constant. On the other hand, the time for a software approach in performing an allocation or a sweeping cycle is non-deterministic. As stated earlier, Java applications spend about 20% of the execution time in dealing with automatic dynamic memory management. This extensive execution time can be greatly reduced with the use of the *DMMU*.

3. Internal architecture of the DMMU

Inside the *DMMU*, three bit-vectors are used to keep all of the object relevant information such as allocation status of the heap, the size information of occupied blocks and free blocks, and the live object pointers. The allocation status is kept on the *Allocation bit-vector* (*A bit-vector*). When a *h_malloc* is called, the size information is received by the

Complete Binary Tree (*CBT*). This dedicated hardware unit is responsible for locating the first free memory chunk that can satisfy the request using the modified buddy system. Besides locating the memory chunk, the *CBT* also has to send out the address of that newly allocated memory and updates the status of that memory block from free to allocated. It is worth noting that while the free block lookup is done using size index of 2^n, the system only allocates the requested size. For example, if 5 blocks of memory is requested, the system will have to find the first free chunk of size 8 (2^3). After a chunk is located, the system only allocates 5 blocks and relinquishes the remaining 3 blocks. Each time an object is created or reclaimed, the *Size bit-vector* (*S bit-vector*) is instantly updated by a dedicated hardware, *S-Unit*. The *auXiliary bit-vector* (*X bit-vector*) is only used during the marking phase of the garbage collection cycle. Once the marking phase is completed, the *sweep* instruction is invoked. A dedicated hardware, *bit-sweeper*, is used to perform this task in constant time. The internal architecture of the *DMMU* is given in Figure 2.

Figure 2. Internal architecture of the *DMMU*.

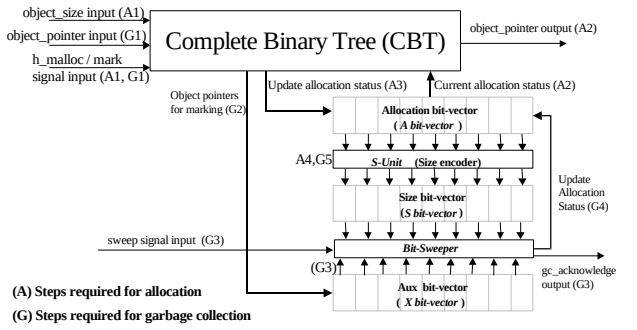


Figure 2 depicts the sequence needed to complete the allocation or garbage collection. For example, if an allocation of size 5 is requested, *A1s* indicate the first step needed to complete the allocation. According the Figure 2, the *h_malloc* and *input signal* would go to logic '1' and the requested size would be given to the *CBT*. Since the *CBT* is a combinatorial hardware, the free memory chunk lookup, the return address pointer, and the new allocation status signals can be produced at the same time (*A2s*). Next, the new allocation status is latched in the *A bit-vector* (*A3*). Since the *S-Unit* is also a combinatorial hardware, as soon as the *A bit-vector* is latched, the new size information is available to the *S bit-vector*. Lastly, the new size information is latched in the *S bit-vector* (*A4*) and the allocation is completed. The sequence of garbage collection can also be traced in a similar fashion.

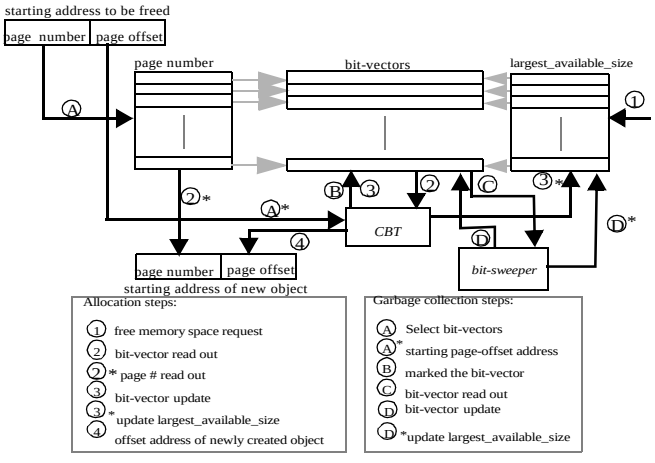
4. Architectural support for *DMMU*

This section summarizes the process of memory allocation and deallocation in the *DMMU*. Since the bit-maps of a given process may be too large to be handled in the hardware domain, the *bit-vector*, a small segment of the bit-map, is used in the proposed system. This idea is very similar to the idea of using *TLB* (*Translation Look-aside Buffer)* in the virtual memory. Due to the close tie between the *S bit-map*, *A bit-map*, and *X bit-map*, the term *bit-vector* used in this section represents one *A bit-vector* (of *A bit-map*), one *S bit-vector* (of *S bit-map*), and one *X bit-vector* (of *X bit-map*). Figure 3 presents the operation of the proposed *DMMU*.

When a memory allocation request is received (step 1), the requested size is compared against the *largest_available_size* of each bit-vector in a parallel fashion. This operation is similar to the tag comparison in a fully associated cache. However, it is not an equality comparison. There is a hit in the *DMMU*, if one of the *largest_available_size* is greater or equal to the request size. If there were a hit, the corresponding bit-vector would be read out (step 2) and sent to the *CBT* [4]. The *CBT* is a hardware unit to perform allocation/deallocation on a bit-vector. For the purpose of illustration, we assume that one bit-vector represents one page of the heap.

After the *CBT* identified the free chuck memory from the chosen page, the *CBT* will update the bit-vector (step 3) and the *largest_available_size* field (step 3*). The object pointer (in terms of page offset address) of the newly created object is generated by the *CBT* (step 4). This page offset combines the page number (from step 2*) into the resultant address.

Figure 3.The allocation and garbage collection processes of the *DMMU*



For the garbage collection, when the *DMMU* receives a mark request, the page number of the object pointer (i.e. a virtual address) is used to select a bit-vector (step A). This

process is similar to the tag comparison in cache operation. At the same time, the page offset is sent to the *CBT* as the address to be marked (step A*). The process is repeated until all the memory references to live objects are marked. When the marking phase is completed, the sweeping phase (step C) would begin by reading out the bit-vectors and send them to the *bit-sweeper*. The *bit-weeper* would keep all of the objects where the starting addresses were provided by step A* and update the bit-vector (step D) and the largest available size field (step D*). The page number, bit-vectors, and the *largest_available_size* are placed in a buffer, called the *Allocation Look-aside Buffer* (*ALB*).

Since the *DMMU* is shared among all processes, content of the *ALB* will be swapped during the context-switching. This issue also exists in *TLB*. To solve this problem, we can add a *process-id* field in the *ALB*. This will allow bit-vectors of different processes to coexist in the *ALB*. We expect the performance of the *ALB* to be very similar to the much-studied *TLB*. However, further research in the *ALB* organization, hit ratio and miss penalty is required.

5. Conclusion

Besides providing the speed and predictability in automatic dynamic memory management, the *DMMU* can also reduce the number of cache misses and page faults. Since all the dynamic memory management information is kept separately from the object, we do not need to bring the object in for the marking and object size look up. The bit-map approach also requires no splitting and coalescing. Additionally, our scheme can also improve the performance of multithreaded applications in a multiprocessor environment. While multithreaded programming in multiprocessor environment promotes parallel execution of threads, the task of managing the heap is still done in a sequential fashion. This means that other threads have to wait if one thread is allocating inside the heap. Since the software approach to allocation is slow and non-deterministic, dynamic memory management can be a major bottleneck in multiprocessor-multithreaded applications that are memory intensive [3]. Our scheme allows allocation to be done quickly, and thus, reduces wait time.

The adoption of object-oriented languages such as C++ and Java in embedded system development also increases the need for a high-performance automatic dynamic memory manager. Industry observers predict that by year 2010, there will be 10 times more embedded system programmers than general-purpose programmers [2]. This prediction is also confirmed by the surge of interests in the web-appliances where each device is a small object-oriented embedded system. In this paper, we introduce hardware instruction extensions that would allow *ADMM* to be fast, robust, and can respond to the hard real-time requirement.

6. References

[1] E. Armstrong, "Hotspot, A new breed of virtual machine", *JavaWorld*, March 1998.

[2] R.W. Atherton, "Moving Java to the factory", *IEEE Spectrum*, December 1998, pp 18-23.

[3] D. Haggander and L. Lundberg, "Optimizing Dynamic Memory Management in a Multithreaded Application Executin on Multiporcessor", *Proc. 1998 Int'l Conference on Parallel Porcessing*, pp. 262-269.

[4] M. Chang and E. F. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems," *IEEE Transactions on Computers*. March, 1996. pp. 357-366.

[5] K. Kavi, J.C. Browne, and A. Tripathi, "Computer Systems Research: The pressure is on" *Computer*, January 1999, pp. 30-39.

[6] R. Jones, R. Lins, *Garbage Collection: Algorithms for automatic Dynamic Memory Management*, John Wiley and Sons, 1996, pp.20-28, 87-95, 296

[7] D. Patterson and J. Hennessy, "Computer Architecture, A Quantitative Approach", Morgan Kaufmann Publishers, Inc., Second Edition 1996.

[8] K. Nilsen and W. Schmidt, "A High-Performance Hardware-Assisted Real-Time Garbage Collection System, *Journal of Programming Languages*, January 1994, 1 - 40.

[9] Paul Wilson, M. Johnstone, M Neely and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review", Proc. 1995 *Int'l workshop on Memory Management*, Scotland, UK, Sept. 27-29, 1995.

[10] Benjamin Zorn, "Custo-Malloc: efficient synthesized memory allocators," Technical Report CU-CS-602-92, Computer Science Department, University of Colorado, July 1992.

Session 2

Architectural Issues in Dynamic Translation

How can hardware support Just-In-Time compilation?

A. Murthy, N. Vijaykrishnan and A. Sivasubramaniam
Department of Computer Science and Engineering
Pennsylvania State University
University Park, Pennsylvania
{amurthy,vijay,anand}@cse.psu.edu

Abstract

Just-In-Time (JIT) compiler is an efficient and preferred style of implementing the Java Virtual Machine (JVM) on resource-rich machines. Using a JIT compiler, the bytecodes are translated to native code at runtime. In this paper, we investigate where the time is spent in such dynamic compilers and how well a smart JIT compiler can do. Next, we propose architectural mechanisms that can be used to support the dynamic code generation and installation.

1 Introduction

The Java Virtual Machine (JVM) [1] is the corner stone of Java technology epitomizing the "write-once run-anywhere" promise. It is expected that this enabling technology will make it a lot easier to develop portable software and standardized interfaces that span a spectrum of hardware platforms. Java programs are translated into a machine-independent JVM format (called bytecodes), to insulate them from the underlying machine architecture on which they would eventually execute. These bytecodes can be executed by: interpretation, Just-In-Time (JIT) compilation [2] or by direct execution on Java processors [3, 4]. Among these techniques, the JIT compiler technology is a prefered style of JVM implementation on resource-rich machines. This paper presents possible directions that one can take in providing architectural support for dynamic compilation.

The rest of this paper is organized as follows. In the next section, we investigate on where the time is spent in JIT compilation. Architectural support mechanisms for enhancing the performance of JIT compilation are described briefly in the section 3. Concluding remarks are provided in section 4.

2 When or Whether to Compile JIT?

Dynamic compilation has been popularly used [2, 5] to speed up Java executions. This approach avoids the costly interpretation of JVM bytecodes, while sidestepping the issue of having to pre-compile all the routines that could ever be referenced (from both the feasibility and performance angles). Dynamic compilation techniques, however, pay the penalty of having the compilation/translation to native code falling in the critical path of program execution. Since this cost is expected to be high, it needs to be amortized over multiple executions of the translated code. Otherwise, performance can become worse than when the code is just interpreted. Knowing when to dynamically compile a method, or whether to compile at all, is extremely important for good performance. Most of the currently available execution environments, such as kaffe [8] and JDK 1.1 [6] employ limited heuristics to decide on when (or whether) to compile JIT. For example, Kaffe typically translates a method on its first invocation, regardless of how long it takes to interpret/translate/execute the method and how many times the method is invoked. JDK 1.2 uses limited heuristics such as detecting presence of loop structure s, while the sophis-

ticated optimizations done by the hot spot compiler announced recently [7] are not yet available. It is not clear how well one could do with a smarter heuristic than what many of these environments provide. We investigate these issues in this section using five specJVM98 [9] benchmarks (together with a simple HelloWorld program[1]) on the Kaffe [8] environment.

Figure 1 shows the results for the different benchmarks. All execution times are normalized with respect to the execution time taken by the JIT compilation mode on Kaffe [8]. For each application, the first bar indicates the time taken to run the program in this mode. This bar is further broken down into its two components, the total time taken to translate/compile the invoked methods and the time taken to execute these translated (native code) methods. The considered workloads span the spectrum, from those in which the translation times dominate such as *hello* and *db* (because most of the methods are neither time consuming nor invoked numerous times), to those in which the native code execution dominates such as *compress* and *jack* (where the cost of translation is amortized over numerous invocations). On top of the JIT compilation execution bar is given the ratio of the time taken by this mode to the time taken for interpreting the program using Kaffe VM. As expected, we find that translating (compiling JIT) the invoked methods significantly outperforms interpreting the JVM bytecodes.

The JIT compilation mode in Kaffe compiles a method to native code on its first invocation. We next investigate how well the smartest heuristic can do, so that we compile only those methods that are time consuming (the translation/compilation cost is outweighed by the execution time) and interpret the remaining methods. This can tell us whether we should strive to develop a more intelligent heuristic at all, and if so, what is the performance benefit that we can expect. Let us say that a method i takes I_i time to interpret, T_i time to translate, and E_i time to execute the translated code. Then, there exists a crossover point $N_i = T_i/(I_i - E_i)$, where it would be better to translate the method if the number of times a method is invoked $n_i > N_i$, and interpret it otherwise. We assume that an oracle supplies n_i (the number of times a method is invoked) and N_i (the ideal cut-off threshold for a method). If $n_i < N_i$, we interpret all invocations of the method, and otherwise translate it on the very first invocation. The second bar for each application shows the performance with this oracle in Figure 1, which we shall call *opt*. It can be observed that there is very little difference between the naive heuristic used by Kaffe and *opt* for *compress* and *jack* since most of the time is spent in the execution of the actual code anyway (very little time in translation or interpretation). As the translation component gets larger (applications like *db*, *javac* or *hello*), the *opt* model suggests that some of the less time-consuming (or less frequently invoked) methods be interpreted to lower the execution time. This results in a 10-15% savings in execution time for these applications. It is to be noted that the exact savings would definitely depend on the efficiency of the translation routines, the translated code execution and interpretation.

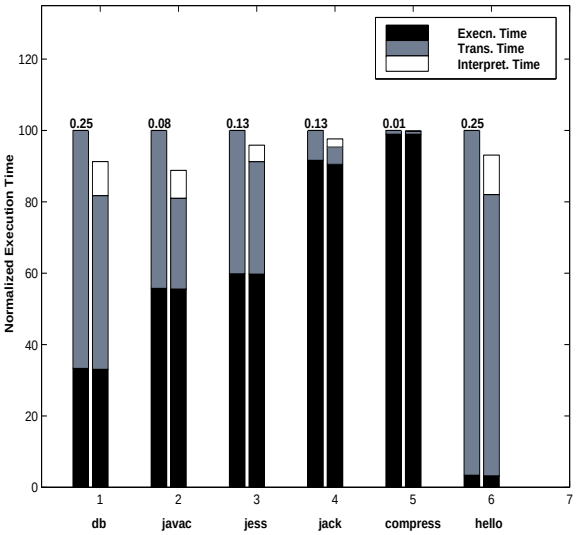


Figure 1: Dynamic Compilation: How well can we do ?

[1] While we do not make any major conclusions based on this simple program, it serves to observe the behavior of the JVM implementation while loading and resolving system classes during system initialization.

The *opt* results give useful insights. Figure 1 shows that regardless of the heuristic that is employed to decide on when/whether to compile JIT, one can at best hope to trim 10-15% in the execution time. On the other hand, we find that a substantial amount of the execution time is spent in translation and/or executing the translated code, and there could be better rewards from optimizing these components. Such optimizations can use better compilation techniques and/or hardware support for dynamic compilation. The next section of this paper discusses architectural support for enhancing JIT compiler performance.

3 Architectural Support for Dynamic Translation

Architectural support for dynamic translation can target at optimizing either the translation part or at improving the performance of the execution part of the translated code. A list of possible directions in providing architectural support for dynamic translation is given below.

- Provide support for removing the dynamic translation of the code from the program's critical path. It may be worthwhile to invest in hardware support (either on the same CPU or using another processor if it is a SMP) that can do the translate in parallel with the interpretation of bytecodes or other useful work done by the actual processor.
- Provide architectural enhancements to support dynamic code generation and installation. While current processors provide facilities to flush the data and instruction caches to support self modifying code, it may be beneficial to provide code generation buffers or a capability to write into I-caches. Such buffers can prevent the translation routines from affecting the locality in the data caches.
- Provide architectural support for profiling. For example, a counter could track the number of hits associated with an entry in the branch target buffer. When the counter saturates, it can trigger the compiler to perform code inlining optimization that can replace the indirect branch instruction with the code of the invoked method. Of course, we may need some mechanism to monitor the program behavior changes to undo any optimizations that may become invalid later. Also, it would be worthwhile investigating the positioning of the translated code towards improving the locality during subsequent execution. Again dynamic monitoring capabilities such as maintaining path history of the method executions can help.
- Provide the capability to extend the dynamic compilation capability to encompass hardware configuration as shown in Figure 1(b). The idea of dynamic configuration is to translate the bytecode sequences of a Java method into reconfigurable hardware on-the-fly. Subsequent invocations of a method pass the data to the reconfigurable hardware; computations are performed in the configured hardware and results are sent back to the calling method.

In the rest of this paper, we will discuss the motivation and mechanism for supporting dynamic code installation and dynamic configuration briefly.

Dynamic Code Generation and Installation Support: In order to investigate whether the translation routines affect the cache locality, we isolated the cache behavior during the translation part and the rest of the JIT compiler execution. The study was performed using the *cachesim5* cache simulator from Shade [10] tool set on an UltraSPARC machine running Sun OS 5.6. The cache behavior of the translate portion is illustrated in Figure 3. The data cache misses in the translate portion of the code contribute to 40-80% of all data misses for many of the benchmarks. Among these, the data write misses dominate within the translate portion and contribute to 60% of misses during translate (see the third bar for each benchmark in Figure 3). Most of these write misses were observed to occur during the generation and installation of the code. Since, the generated code for the method is written to memory for the first time, it results in compulsory misses in the D-Cache. One may expect similar compulsory

misses when the bytecodes are read during translation. However, they are relatively less frequent than the write misses since 25 native (SPARC) instructions are generated per bytecode on an average [11].

Installing the code will require writing to the data cache, and these are counted as misses since those locations have not been accessed earlier (compulsory misses). These misses introduce two kinds of overheads. First, the data has to be fetched from memory into the D-cache before they are written into. This is a redundant operation since the memory is initialized for the first time. Second, the newly written instructions will then be moved (automatically on instruction fetch operations) from the D-cache to the I-cache. It would be useful to have a mechanism wherein the code can be generated directly into the I-cache. This would require support from the I-cache to accommodate a write operation (if it does not already support it), and preferably a write-back I-cache. It should also be noted that for good performance, one should be careful to locate the code for translation itself such that it does not interfere/thrash with the generated code in the I-cache.

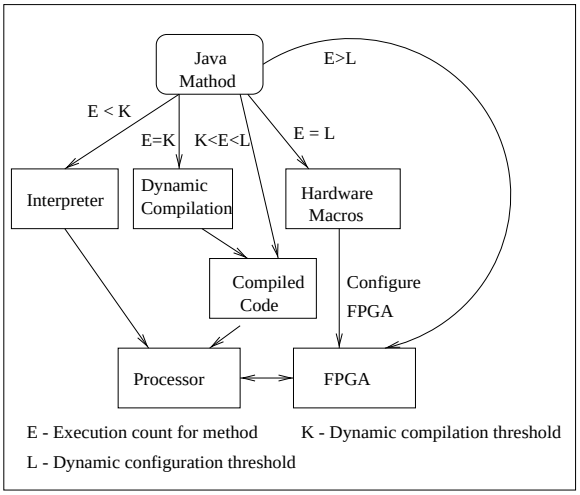


Figure 2: A Dynamic Translator with Dynamic Configuration and Compilation Capability

Dynamic Configuration: Figure 1 shows that there are applications, like *compress*, and *jack*, in which a significant portion of the time is spent in executing the translated code. There is a common (and interesting) trait in these applications, where the execution time dominates and a significant portion of this time is spent in certain specific functions. For instance, *compress* employs a standard set of functions to encode all the data. If one optimizes the execution of such functions, then there is hope for much better performance. Hardware configuration of these methods using Field Programmable Gate Arrays (FPGA) on-the-fly (similar to how JIT compiler dynamically opts to compile-execute rather than interpret) is an interesting option.

We are currently modifying the dynamic compiler to integrate the dynamic configuration mechanism as shown in Figure 2. We plan to use hardware (objects) cores corresponding to selected Java methods developed by hardware designers with a specific interface to the rest of the JVM as opposed to dynamically translating bytecodes into a FPGA configuration file [12]. This alternative can avoid the hardware compilation cost. It also benefits from the better performance of a customized hardware core over a synthesized (hardware compiled) core. When a method is executed for more times than a predefined threshold, the existence of a hardware core for the requested method is checked. If a core is found, the dynamic translator initiates the loading of the configuration file of the corresponding hardware core into the FPGA. The execution of the method proceeds in a parallel thread either in interpreted or JIT compiler mode during the configuration. When a subsequent call is made to the same method, the completion of the configuration process is checked. If the configuration is complete, the input data to the method is passed to the FPGA and the execution begins in the configured hardware. When the method is executed on the FPGA, the JVM could either be busy polling until the results are returned or switch to the execution of another thread. The output data generated by the customized hardware is buffered and transferred to the processor once the execution is complete. We could also use the partial reconfiguration capability in the FPGAs to implement multiple cores for supporting different methods simultaneously.

4 Conclusion

The key to an efficient Java virtual machine implementation is the synergy between well-designed software, an optimizing compiler, supportive architecture and efficient runtime libraries. This paper has looked at only a small subset of issues with respect to supportive architectural features for Java, and there are a lot of issues that are ripe for future research.

References

[1] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[2] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, " Compiling Java just in time ," *IEEE Micro*, vol. 17, pp. 36–43, May-June 1997.

[3] H. McGhan and M. O'Connor, " PicoJava: A direct execution engine for Java bytecode ," *IEEE Computer*, pp. 22–30, October 1998.

[4] N. Vijaykrishnan, *Issues in the Design of a Java Processor Architecture*. PhD thesis, College of Engineering, University of South Florida, Tampa, FL 33620, July 1998.

[5] U. Holzle, "Java on Steroids: Sun's high-performance Java implementation," in *Proceedings of HotChips IX*, August 1997.

[6] "Overview of Java platform product family ." http://www.javasoft.com/products /OV_jdkProduct.html.

[7] D. Griswold, "The Java HotSpot Virtual Machine Architecture ," March 1998. Sun Microsystems Whitepaper.

[8] " Kaffe Virtual Machine." http://www.transvirtual.com.

[9] " SPEC JVM98 Benchmarks." http://www.spec.org/osg/jvm98/.

[10] R. F. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," Tech. Rep. SMLI TR-93-12, Sun Microsystems Inc, 1993.

[11] R. Radhakrishnan, J. Rubio, and L. John, "Characterization of Java applications at the bytecode level and at UltraSPARC-II Machine Code Level," in *Proceedings of International Conference on Computer Design*, October 1999. To appear.

[12] J. M. P. Cardoso H. C. Neto, "Macro-based hardware compilation of Java bytecodes into a dyna mic reconfigurable computing system", in Proceedings of the 7th IEEE Symposium on Field Programmable Custom Computing Machines, April 1999.

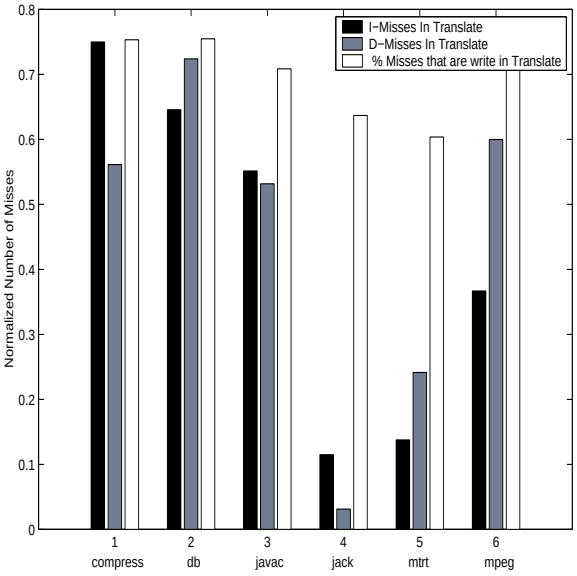


Figure 3: Cache Misses within Translate Portion. Cache configuration used : 4-way set associative, 64K DCache with a line size of 32 bytes and 2-way set associative, 64K ICache with a line size of 32 bytes. The first bar shows the I-Cache misses in translate relative to all I-Cache misses, the second bar shows the D-Cache misses in translate relative to all D-Cache misses, the third bar shows the D-cache write misses in translate relative to overall D-cache misses in translate.

Exploiting Hardware Resources: Register Assignment across Method Boundaries

Ian Rogers, Alasdair Rawsthorne, Jason Souloglou
The University of Manchester, England
{Ian.Rogers,Alasdair.Rawsthorne,Jason.Souloglou}@cs.man.ac.uk

Abstract

Current microprocessor families present dramatically different numbers of programmer-visible register resources. For example, the Intel IA32 Instruction Set provides 8 general-purpose visible registers, most of which have special-purpose restrictions, while the IA64 architecture provides 128 registers. It is a challenge for existing code generators, particularly operating within the constraints of a just-in-time dynamic compiler, to use these varying resources across a number of architectures with uniform algorithms. This paper describes an implementation of Java using Dynamite, an existing Dynamic Binary Translation tool. Since one design goal of Dynamite is to keep semantic knowledge of its subject machine localized to a front-end module, the Dynamite code generator ignores method boundaries when allocating registers, allowing it to fully exploit all hardware register resources across the hot spots of a Java program, regardless of the control graphs represented.

1. Introduction

Current microprocessor families present dramatically different numbers of programmer-visible register resources. For example, the Intel IA32 Instruction Set [1] provides 8 general-purpose visible registers, most of which have special-purpose restrictions, while the IA64 architecture [2] provides 128 registers. In the former case, register renaming and out-of-order issue is used in the microarchitecture to exploit a richer resource set than indicated by the instruction set, but the paucity of the visible instruction set remains a severe constraint on a just-in-time code-generator. In particular, it is difficult to pass method parameters efficiently in registers in x86 implementations, since register pressure ensures that the lifetimes of values in registers are so short.

In the case of RISC, and particularly EPIC [3] architectures, the visible instruction set more closely models the register hardware resources provided in an implementation. Register assignment becomes viable using a conventional approach, such as defining a method calling sequence, involving caller-saved, callee-saved and parameter registers, and allocating local variable and temporary registers within individual methods.

In the distant future, it is possible that unconventional CPU architectures may provide extremely high levels of performance without any significant number of registers.

This disparity of hardware resources has been addressed by designers of current Java Virtual Machines by providing different register allocation algorithms for (e.g.) IA32 and RISC machines. In this paper, we describe a single register allocator appropriate to register-rich and register-poor architectures, and we explain how this allows us to set optimization boundaries independently of method boundaries, giving performance advantages.

Conventional static compilers, and existing JIT compilers, use method-inlining, more or less aggressively, to uncover a number of optimization possibilities, with register assignment among them. Inlining may have its limitations, however, and we have found a number of cases where inlining (particularly leaf-method inlining) does not completely address hot regions of an application.

In this paper, we present some techniques we use to run Java byte-coded programs on Dynamite, our existing dynamic binary translation environment. As will be described below, our techniques rely on optimizing regions of code without reference to the semantic boundaries of methods. Broadly, we claim to gain the performance benefits of inlining, without its limitations.

We describe the Dynamite binary translation system, its interfaces and its approach to optimization. In section 4, we show how Java programs are implemented using the Dynamite facilities, and section 5 discusses our preliminary results. Previous work in register assignment for Java programs is introduced in section 6.

2. Dynamite

Dynamite is a reconfigurable Dynamic Binary Translation system, whose aims are to extend the "Write Once, Run Anywhere" paradigm to existing binary applications, written and compiled for any binary platform. To enable the quick configuration of a particular translator, Dynamite is constructed as a three-module system, as shown in figure 1.

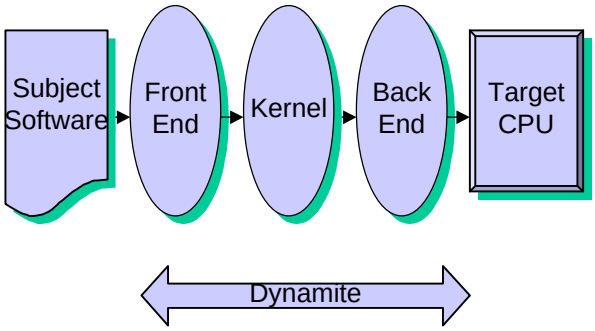


Figure 1. Dynamite Structure

The function of the major components is almost self-explanatory: the Front End transforms a binary input program into an intermediate representation (IR), which is optimized by the Kernel, and the Back End generates and executes a binary version on the target processor. The Front End interface supports a number of abstractions convenient for efficient Front End implementation, such as the "abstract register", which holds intermediate representations of the effects of subject instructions. This interface is configured by an individual Front End module, since different subject architectures require different numbers of registers.

Two features of this front end interface are relevant to the current discussion: firstly, the interface resembles a RISC-like Register Transfer Language, containing no peculiarities (such as condition codes or side-effects) adapted for particular subject architectures. Secondly, procedure (method) calling is achieved at a primitive level, typically by having the front end create IR to compose a link value in subject state space, and then branching or jumping to the callee. Return from a procedure is similarly implemented by having the Front End create IR which causes a jump to be made to the return address. Parameter passing and stack-frame management is implemented by having the Front End create the appropriate IR to model the subject architecture's requirements.

The Kernel contains about 80% of the complexity of the translator system. It creates and optimizes IR in response to Front End calls, and invokes the Back End to code generate and execute blocks of target code. Register assignment for the target machine code generator currently takes place within the Kernel, again using a Back End interface that is parameterized for specific target architecture. Optimization is performed adaptively at a number of different levels, starting with initial translation as described below.

To achieve its performance goals, Dynamite operates in an entirely lazy manner. An instruction is never translated until that instruction must be executed, either because it is a control (jump, branch, exception or call) target, or because the immediately preceding branch has fallen through. As instructions are decoded by the Front End, their IR is combined until a control transfer is encountered. During this process, the kernel performs optimizations such as value forwarding and dead code elimination. When the block of IR is complete, it is code generated, executed by the back end, and cached for subsequent reuse. After a block of target code is executed, its successor location may either be found within the cache, or may need translation using the same actions.

In efficiency terms, target code blocks generated using this initial scheme leave something to be desired. The benefit is that the initial translation is quick, taking only a few thousand instructions per subject instruction. Register usage is determined by an individual Back End, largely as a result of the method calling sequence mandated by the static compiler used to compile Dynamite. Target registers are used to store temporary results within the block, but existing Back Ends preserve all subject register values in target memory at the boundary of basic blocks.

More optimization and higher quality code generation are triggered when an individual target block is executed more frequently than a dynamic execution threshold. This event causes the kernel to create a group containing this and related blocks in a hot region, and to optimize this Group Block as a single entity. Group Blocks may span arbitrary boundaries in the subject machine: indeed, in other applications, Dynamite optimizes across programs and their procedures, static and dynamically-linked libraries, OS Kernels, and across different virtual machines.

Within a Group Block, the Dynamite kernel examines the existing control flow of the region, identifying certain blocks as entry and exit blocks, and performing value propagation and dead-code elimination across the entire group. The control flow is used to straighten the conditional branches and eliminate jumps within the group, so that frequent cases fall through, minimizing taken branches and maximizing I-cache utilization.

Code generation for a Group Block occurs next. To avoid the expense of an iterative algorithm, a very simple incremental register allocation algorithm is used. Starting with the target block, operands are allocated to registers (if the target machine architecture requires), and operation results are allocated to registers if they are to be reused. As the register set is exhausted, spill code is generated to relinquish previously allocated registers for new operations. Register allocations are carried across basic block boundaries, and the act of code generating from the most- to the least-frequently executed blocks within the group ensures that spills are minimized.

We emphasize that during this code generation process, all abstract registers and target registers are treated symmetrically. We do not distinguish between

registers used to pass parameters, those used to carry visible results, and those used to hold temporary values. In this way, we can code generate a region containing multiple procedure call and returns as efficiently as one containing just a portion of a large procedure.

The final stage of code generation is to generate stubs for the entry and exit blocks, which need to load abstract register values into target registers and compute and store exit values from the group block.

This group-block creation phase can be invoked and re-invoked any number of times during program execution, creating larger and smaller groups of basic blocks, always independent of method boundaries, as the subject program proceeds through its execution.

3. Implementing Java

The critical design decisions when implementing Java using Dynamite are the mapping of JVM registers, local variables, and the stack to the relevant Dynamite objects, namely abstract registers .

To allow Dynamite to optimize across different Java methods, we need to map multiple stack frames simultaneously to different abstract registers. Two schemes were considered for doing this.

3.1 Sliding frame

On entering a method the front-end would create a frame within the abstract registers. The arguments to the method (stored at JVM local variable 0 upwards) become the base for the frame. After the local variables the return address is held in the next available abstract register. The JVM stack is held at the end of the frame. However, studies [4] show that the stack is usually empty on basic block boundaries. The purpose of the stack in the frame is therefore to hold onto stack values that occasionally span basic blocks and to pass arguments to called methods. The arguments could become part of the next frame by overlapping the stack part of the caller's frame and the local variable part of the callee frame.

Unfortunately, the problem with this scheme is that the IR for an individual method needs to refer to specific abstract registers. This fixes its translation to a particular stack depth. If the same method is called at a different stack depth, we need to re-translate it for this new depth. This is particularly expensive for recursive methods. We could possibly generate special case translations for recursive methods and fall back on a scheme that saves the frame to memory on a method call. Otherwise, for methods that are called from multiple stack locations we could avoid re-translation if the method's frame is at a greater abstract register location than the current frame. We would, however, still have to copy the arguments to the method from the caller's frame to that of the callee.

Our research has shown that around 90% of execution time is spent in methods called from more than 16 call sites. These methods are typically utility functions which are prime candidates for optimization. Expensive optimizations would be prohibitive for these methods as the optimization would need repeating many times.

We conclude that using a sliding frame is therefore undesirable.

3.2 Fixed frame

The drawback with the "sliding-frame" scheme is that it is necessary to recompile methods called with different frame base pointer values. If we fix the address where a method's frame lives in abstract registers we remove this problem. To do this, we allocate a new, unique frame from a large pool of abstract registers the first time a particular method is invoked.

We do, however, still need to pass arguments to the called method from the stack of the calling method. On encountering a method call, the arguments to the method are held in intermediate representation ready to be written to registers. At this point, they can be written directly to the called method's local variables avoiding any copy operation.

The first penalty for this scheme is that we need to retranslate these abstract register assignments for different methods called from the same call site. A study using Harissa [5] shows that at least 40% of method calls can be accurately statically predicted for 100% of the time, and dynamic statistics are even better than this.

As in the "sliding-frame" design, recursion needs to be handled differently, as two invocations of a method cannot share a single frame. A simple scheme to handle recursion is to save a frame to memory before using it, if it is active, and to restore it on exit. Alternatively, we may find some circumstances in which it is advantageous to generate special-cased versions of recursive methods, each of which uses a different frame of abstract registers. This special-casing will be triggered by a heuristic monitored by code planted in the initial translation of a (potentially recursive) method.

Finally, assigning unique abstract registers to every method presents a problem when the static pool is exhausted. For programs studied to date, fewer than 8000 abstract registers would be sufficient. If greater numbers were required, the front-end could start re-using frames. For example, all leaf methods can share the same frame, and more generally, methods that occur only on disjoint subtrees of the call graph can share frames. In the pathological case, frames of abstract registers can be reused by planting code that spills a number of frames to memory and refills them when necessary.

4. Discussion

To evaluate this scheme before its implementation, we carried out a number of experiments by instrumenting

Kaffe [6] to log information about the dynamic behaviour of Java applications. We keep sufficient information to create the dynamic method call tree of the application. We create a call tree, in which methods appear once for each call site, to assess our implementation alternatives. For each method occurrence, we keep the number of byte codes executed in this invocation, and its local variable requirement, including parameters.

To estimate the number of target registers needed in optimization regions of different sizes, we identify hot spots on this call tree (methods with high contributions to overall instruction counts), and successively add them to optimization regions, counting the total number of local variables required at each step. This approximates to code generating by hottest method first, then by successively cooler region. As each successive method is added to the optimization region, we require more local variables. This gives us the characteristics we show below. For these experiments, we monitored "javac", the Java compiler in Java, since it is the largest Java application we can find.

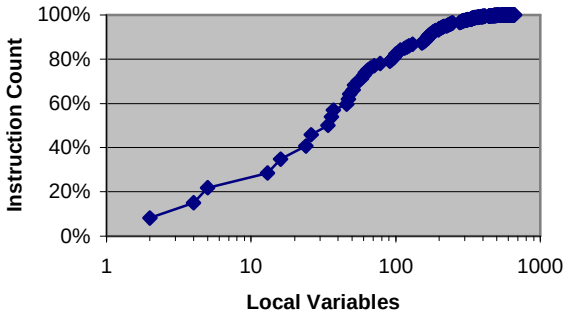


Figure 2. Instruction Count against Local Variables

In figure 2, we show how a straightforward "hottest first" selection algorithm can use varying numbers of target registers to code generate "spill-free" regions of different sizes. As we intuitively expect, as we allocate more and more local variables into target registers, we can encompass larger and larger regions, contributing to ever increasing fractions of total instruction count. For example, with 5 target registers, we can code generate a region that contributes 22% to total execution count, and with 26 registers, 46% of execution count.

In Figure 3, we use a slightly different heuristic to select methods to include within our selection region. Here, we select methods based on their run-time contribution per local variable. That is, comparing methods with similar run-time contributions, we

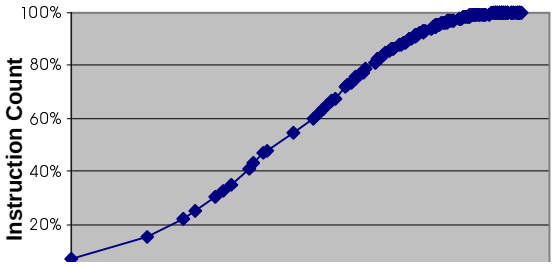


Figure 3. Instruction Count against Local Variables

Local Variables

preferentially select the method with the smaller requirement for local variables. This gives better results when there are fewer target registers available: for example with 8 registers, we can cover 30% of total instruction count, and with 25 registers, we can cover 54% of the instructions.

5. Previous Work

The success of Java has resulted in many JVM implementations. Some implementations such as Harissa [5], and J2C translate Java to C code. They then rely on a C compiler to perform register allocation within and over method call boundaries. Register mappings within C programs are beyond the scope of this paper.

In this section we examine how other JVM implementations perform register mapping and allocation and compare these to Dynamite.

5.1 Register allocation

Cacao [4] initially maps the JVM stack and local variables to pseudo-registers, which are then allocated to CPU registers. Each mapping and allocation begins at the start of a basic block and builds on the mappings and allocations of previous basic blocks. When CPU registers are exhausted a register is spilled to memory and filled by a pseudo register.

On method call boundaries Cacao pre-allocates registers. It uses CPU registers to pass arguments and to receive return values. On machines without register windows pre-allocation of arguments is only possible for leaf methods.

Pre-allocation can tie in with existing compiler method call conventions: for example, in the DAISY JVM [7] arguments and return values are passed and received using the Power PC's C compiler calling conventions, which uses standard registers for passing arguments.

5.2 Comparison with Dynamite

Method invocation creates a new frame on a call stack. Cacao avoids unnecessary accesses to this frame by pre-allocation, utilizing register windows or potentially by using the machine's standard calling convention. However, these static mappings take no account of run-time information on register usage. Therefore registers could be allocated and then subsequently unused. Cacao would also calculate any parameters even if they were unused. Cacao would also have to copy from one register to another if it repackaged arguments to another method. Dynamite on the other hand can avoid this by value forwarding and dead code elimination within a group block.

Also, when registers are spilled only the surrounding and previous basic blocks are considered. This means that a pseudo register could be spilled in one basic block and then filled back again in the next, and Cacao wouldn't know it could spill different registers which are unused in subsequent basic blocks. Dynamite's runtime information about register usage can provide a better register allocation in this case.

6. Conclusion

In this paper, we have introduced Dynamite, an environment for creating dynamic binary translators. We have shown how the run-time concepts of the Java Virtual Machine are mapped onto the Dynamite front end interface and its internal register allocation algorithms. This mapping necessarily discards the concepts of methods and their local variables.

Preliminary investigations show that "method-free" register allocation shows promise for efficient code generation across architectures providing wide ranges of hardware register resources. We look forward to presenting more definitive numerical results at the workshop in October.

7. References

[1] Intel Corporation, "Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture," Order Number 243190, 1999.

[2] Intel Corporation, "IA-64 Application Developer's Architecture Guide," Order Number 245188, 1999.

[3] Trimaran consortium, "Trimaran Project Homepage," http://www.trimaran.org

[4] Andreas Krall, "Efficient JavaVM Just-in-Time Compilation," *International Conference on Parallel Architecture and Compilation Techniques (PACT98)*, Paris, France, October 13-17, 1998.

[5] G. Muller, B. Moura, F. Bellard, C. Consel, "Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code," *Third USENIX Conference on Object-Oriented Technologies (COOTS-97)*, Portland, Oregon, June 16-20 1997.

[6] Transvirtual Technologies Inc., "Kaffe Product Architecture," http://www.transvirtual.com/products/architecture.html

[7] K. Ebcioglu, E.R. Altman, E. Hokenek, "A JAVA ILP Machine Based on Fast Dynamic Compilation", *IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java*, Eilat, Israel, January 9-10, 1997.

A Decoupled Translate Execute (DTE) Architecture to Improve Performance of Java Execution

Ramesh Radhakrishnan and Lizy Kurian John
Laboratory for Computer Architecture
Department of Electrical and Computer Engg.
The University of Texas at Austin, Austin, Texas 78712
{radhakri,ljohn}@ece.utexas.edu

Abstract

Java is increasing in popularity in the software industry, and is being used to implement server and enterprise scale projects. Such applications require high performance, and need to run efficiently. Techniques like JIT compilers and PicoJava chips offer some speedup, but in this paper we look at alternate techniques in hardware which can be incorporated in future microprocessors. We propose a Decoupled Translate Execute (DTE) Architecture, which takes advantage of run-time characteristics of Java. The DTE architecture can be implemented using a mix of hardware and software or alternatively purely in hardware.

1 Introduction

Java technology has been adopted by the enterprise to offer software solutions in various fields, due to its advantages of portability, security and modularity. A factor which affects the execution speed of Java is that it is still predominantly a software emulated language. A program written in Java is translated (by software emulation) to an architecture-neutral instruction set called bytecodes. These bytecodes are platform independent and can be executed on any system which supports the Java runtime environment. Typically the bytecodes are interpreted or converted to native instruction set of the machine by a Just-in-Time (JIT) compiler.

Translation consumes a major part of the cycles during Java execution using a JIT compiler [1]. In many of the SpecJVM98 programs, approximately half the time is spent in compile time operations and only half of the time is spent in execute related operations [1]. The data cache miss rates during execution with a JIT compiler were seen to be dominated by compulsory write misses, which were arising from installation of the translated code. Another phenomenon is that the translated code is deposited into the data cache and later fetched from the instruction cache, results in avoidable data transfer and double-caching. In this paper, we present the Decoupled Translate Execute (DTE) Architecture which forms a solution to both of the above problems.

2 The DTE Architecture

Translation and *Execution* are two distinct operations occurring during Java execution. While interrelated, there is significant potential overlap that can be exploited between these two operations using a Decoupled Translate Execute (DTE) Architecture along the lines of the traditional DAE architectures [2, 3, 4]. Figure 1 illustrates the structure of the proposed decoupled architecture, which consists of the execute and translate processors and necessary buffers for interprocessor communication. The bytecode executable will enter the Execute Processor (EP) and the execute processor will check whether this method is available in the Translated Code Cache (TCC) in the translated form. If it is not found in the TCC, it will deposit the methods to be translated into the E-to-T queue. The Translate Processor (TP) will translate the code and put the translated methods into the TCC. The translate operation progresses concurrently with execution, once a few routines have been translated. The execution proceeds as long as the next required method has been translated and deposited into the TCC. The granularity for translation can be bytecode or *method*. Although queue might simplify control, a cache-like structure can help method reuse. Hence we are using a cache-like structure for TP to EP communication and a simple queue structure for EP to TP communication.

Unlike DAEs, splitting code into translate and execute threads is easier and straightforward due to the inherent threaded nature of Java code. The improvement obtained using such a decoupling strategy will

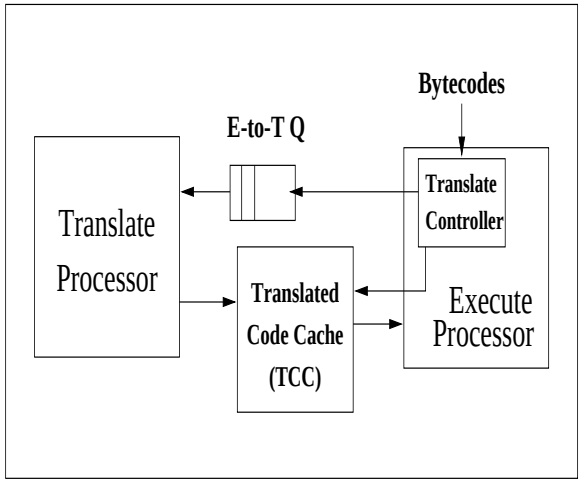


Figure 1: The Decoupled Translate Execute Architecture (DTEA)

E-to-T Q is a FIFO buffer between the Execute processor (EP) and the Translate processor (TP), used by the EP to communicate methods/bytecodes to be translated to the TP. TCC is a cache used by TP to communicate translated methods/bytecodes back to the EP. It stores the translated methods/bytecodes and facilitates method reuse.

depend on the overlap that can be attained between translate and execute operations. In traditional DAE terminology, this would be called *slip*. The distinctly separate threads required during Java execution provides potential for significant overlap and parallelism.

The translate processor may use software translation or hardware translation. Software translation consists of exactly the same operations that happens in current JITs during the translate phase. Hardware translation would involve template matching and generation of translated code as modern x86 processors generate ROPs(RISC ops) or UOPS(RISC like micro-operations). A multithreaded processor utilizing different threads for translation and execution may be considered as a software implementation of the DTEA.

3 The Hard DTE Architecture

In this section we describe a subset of the DTE Architecture, the "Hard DTE" Architecture. In the Hard DTE architecture the translation of bytecodes to native code is done through a hardware translator, as opposed to using a software emulator. This provides advantages of speed in translation, but at the expense of increasing design complexity and chip area.

Software emulation of bytecodes (interpretation, JIT compilation) is slow. Our studies on Java workloads [5] show that interpreted execution requires on the average 35 SPARC instructions to emulate each bytecode. The average SPARC instructions required to emulate a bytecode was approximately 20 when using a JIT compiler [6]. Software emulation is easy to implement for new platforms but cannot offer a solution for fast execution of Java bytecodes. Using a hardware solution, which implies executing the bytecodes directly in hardware would eliminate the requirement of a software layer to emulate the bytecodes.

There are several issues to be considered while employing hardware to perform translation. The current hardware implementations of the Java Virtual Machine (JVM) such as PicoJava are stack based. However, the Instruction Level Parallelism (ILP) which can be exploited using a stack architecture is limited. Figure 2 shows the percentage reduction in execution cycles and improvement in instructions per Cycle (IPC) for Java applications executed using the interpreter and the JIT compiler. The programs executed using the JIT compiler are seen to show slightly higher improvements in both execution time and IPC when executed on 4-way and 16-way processors. The individual data for each benchmark can be found in [1], and we include Figure 2 in this document to serve as a motivation for hardware translation into a non-stack ISA. Our simulator models realistic branch prediction, however, assumes a perfect Branch Target Buffer (BTB) and therefore the performance of the interpreted programs will be still worse on a real machine in which the BTB is modeled (since the interpreter has more indirect branches than the JIT mode of execution).

Based on results of preliminary performance estimates, we favor a hardware implementation of the DTEA concept. Both EP and TP can be built on the same chip and the architecture can be very similar to modern dynamically scheduled microprocessors except for a few building blocks. This architecture will include a hardware unit to convert the bytecodes into simple RISC like instructions. These converted instructions will be register based, thereby enabling use of standard techniques to obtain high performance. The Translated Code Cache (TCC) will store the converted bytecodes. The TCC will be very similar to a Trace cache [7, 8], in the sense that it captures the dynamic translated code during program execution. A high-level block diagram of a system with the proposed hardware units is shown in Figure 3. Instead of fetching bytecodes from the standard instruction cache, whenever possible, the converted bytecodes from the decoded bytecode instruction cache will be directly fetched and executed by the processor core. All blocks in Figure 1, except the highlighted blocks are part of any state of the art micro-

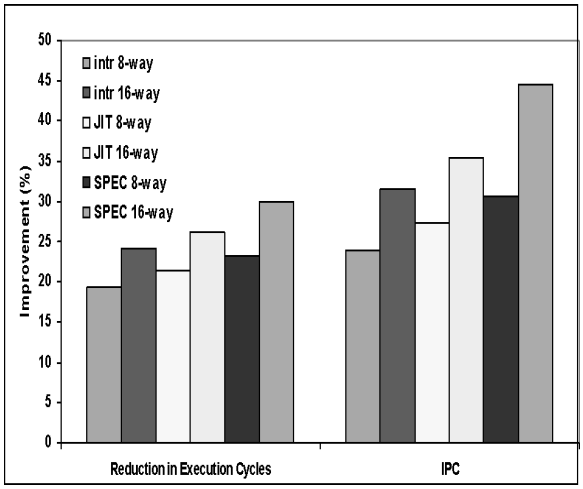


Figure 2: A comparison of scalability during interpreted and JIT compiled execution.

Percentage reduction in execution cycles and improvement in IPC relative to a 4-way configuration is shown. The interpreted code is increasingly stack-oriented, while the JIT code incorporates register optimizations. (This figure is provided as motivation for non-stack ISA in the core of Java processors.)

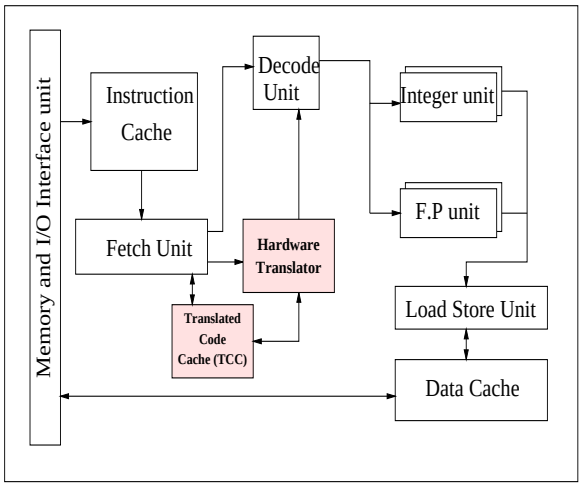


Figure 3: The Hard-DTEA implementation - microarchitecture of a decoupled processor that performs translations in hardware

processor. Although not explicitly indicated, there are queues between various blocks in Figure 3, especially between the fetch unit and the Hardware Translator. Storing the translated bytecodes/methods in the TCC also provides opportunity for a variety of optimizations as performed by the fill-unit in conjunction with a trace cache [9].

3.1 Fill Unit approach to decoding bytecodes

The block diagram of the front-end for the Hard DTE is shown in Figure 4. The hardware translator/decoder will convert bytecodes into smaller fixed length instructions, which will be packed into the Translated Code Cache (TCC) by the fill unit. A line in the TCC can be finalized or completed when one comes across a control flow instruction. The branch unit will provide the next instruction address, and it is fed back to the instruction cache and the TCC. If the lookup in the TCC results in a hit, then the decoded instructions are fed to the dynamically scheduled micro-engine directly from the TCC.

The fill unit and the TCC help in creating larger atomic units of work, which can be fed to the dynamically scheduled micro-engine. Increasing the fetch bandwidth, using the hardware translator and the TCC will allow us to execute the bytecodes in a superscalar

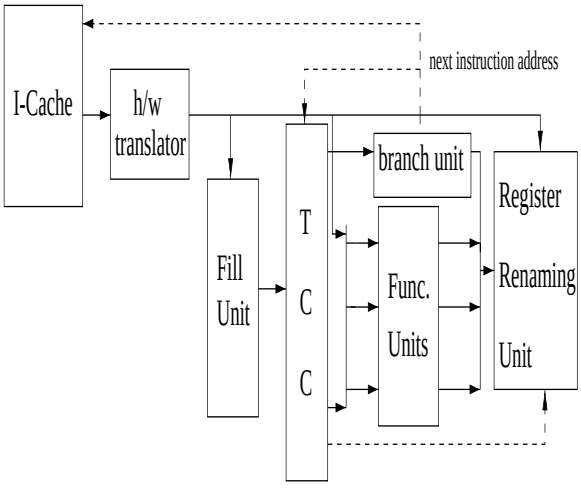


Figure 4: Block diagram of the Hardware DTEA with an instruction fill unit

fashion, unlike the PicoJava which did not execute instructions out of order as the instruction fetch bandwidth was very limited. The PicoJava fetches only 1-2 bytecode instructions every cycle. The stack creates an implicit dependency amongst the bytecodes and the variable length of the bytecode instructions make it harder to decode multiple instructions simultaneously.

3.2 Decoding bytecodes into micro-ops

A hardware decoder will convert the bytecodes to multiple micro-operations. A decoding structure similar to the Intel P6 decoder can be used, which can decode multiple simple and one complex instruction in one cycle. Another approach is to decode only the commonly used bytecodes and store them in the TCC, and to execute the other bytecodes by trapping to micro-code or software.

Consider the bytecode, `dup2`. This bytecode duplicates the top two elements of the stack and pushes them on the stack. This bytecode can be split into the following five micro-operations:

```
TOSB <--  [SP-4] ; load TOS (top of stack)
TOSA <--  [SP-8] ; load TOS-1
SP <-- SP + 8    ; TOS = TOS + 2
[SP-4] <-- TOSA  ; store new TOS
[SP-8] <-- TOSB  ; store new TOS-1
```

In decoding this bytecode, we used 2 temporary registers TOSA and TOSB. We also modified the stack pointer (SP). The bytecode required 2 stores, 2 loads and 1 computation instruction and used 2 intermediate registers to store temporaries. Similarly, we can calculate the load/store instructions and computation instructions required to implement different bytecodes to come up with a resource template which can be used during decoding.

4 Concluding Remarks

In this paper we propose the DTE architecture and also describe a Hard DTE, which is a subset of the DTE architecture. The DTE architecture takes advantage of the concurrency available at Java run time, during the *translate* and *execute* stages.

Other researchers have proposed to take advantage of the concurrency between translate and execute by using a separate thread on a separate processor to do the translation. The DTE architecture differs from such a scheme in the fact that the communication between the TP and EP takes place through Queues and the TCC, which reduces the cost. In the DTE each processor can run ahead of each other when possible, since we use buffers to store the result of each processor. Scheduling threads to run on different processors can be expensive if there is a lot of switching, and unless there is a lot of method re-use we would see a lot of translation taking place. By adding more functionality to the translate processor, we can keep it busy in the case of long running application with high method reuse. The TP can predict or guess the methods which are going to be called in the future and compile them, so that the native code can be used the next time the method is invoked.

The DTEA architecture and in particular, the Hard-DTEA are being evaluated using the SpecJVM98 benchmarks. A set of Java micro-benchmarks are also used for validation purposes in the study. The ability of the DTEA concept to exploit concurrency between translation and execution, hide the cost of translation, and enable Java to achieve the performance of off-line compiled code without losing the advantages associated with JIT compilation is being studied.

References

[1] R. Radhakrishnan, N. Vijaykrishnan, L. John and A. Sivasubramanium, "Architectural Issues in Java Runtime Systems," Tech. Rep. TR-990719, Laboratory of Computer Architecture. Electrical and Computer Engineering Department, University of Texas at Austin, 1999. http://www.ece.utexas.edu/projects/ece/lca/ps/tr990719.pdf.

[2] J. E. Smith, " Decoupled Access/Execute Computer Architecture," in *ACM Transactions Computer Systems*, pp. 289–308, November 1984.

[3] L. John, P. T. Hulina, and L. Coraor, "Memory latency effects in decoupled architectures with a single data memory module," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 236–245, May 1992.

[4] L. John, V. Reddy, P. T. Hulina, and L. Coraor, "Program balance and its impact on high performance RISC architectures," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 370–379, Jan 1995.

[5] R. Radhakrishnan, J. Rubio, and L. John, "Characterization of Java applications at the bytecode level and at UltraSPARC-II Machine Code Level," in *Proceedings of International Conference on Computer Design*, October 1999. To appear.

[6] R. Radhakrishnan, J. Rubio, L. John and N. Vijaykrishnan, "Execution characteristics of just-in-time compilers," Tech. Rep. TR-990713, Laboratory of

Computer Architecture. Electrical and Computer Engineering Department, University of Texas at Austin, 1999.
http://www.ece.utexas.edu/projects/ece/lca/ps/tr990717.ps.

[7] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, Dec 1996.

[8] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[9] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," in *Proceedings of the 31st ACM/IEEE International Symposium on Microarchitecture*, Dec 1998.

Session 3

Object-Oriented Architectural Support

Applying Predication to Reduce the Cost of Virtual Function Calls

Chris Sadler and Sandeep K. S Gupta
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
{sadler,gupta}@cs.colostate.edu

Rohit Bhatia
VLSI Technology Center (VTC)
Hewlett-Packard Company
3404 East Harmony Road
Fort Collins, CO 80528-9599
rxb@fc.hp.com

Abstract

The direct costs of virtual function calls in object-oriented programs is a runtime overhead incurred by the number of operations required to compute a target function address, and the time to perform these operations. We present a technique that uses an HPL PlayDoh architectural feature known as predication to reduce the direct costs of virtual function calls. This technique is based on the possibility that the same virtual function table will be shared between virtual function calls, and whereby exploits this possibility by interleaving the function calls for objects whose type cannot be determined statically. With cost models we show that predication will eliminate redundant loads of the virtual function table from memory, and thereby reduce the impacts of memory latency on the overall runtime performance of virtual function calls.

1. The overhead of virtual function calls

The use of virtual function calls within object-oriented (OO) languages has a direct cost of degrading the runtime performance of programs. As opposed to static function calls that can be resolved during compilation, a virtual function call is resolved during runtime if it cannot be statically bound during compilation. Hence, it will incur a runtime overhead to determine which function entry point to jump to. A study performed by Driesen and Holzle showed that C++ programs spend a median of 5.2% of their execution time, and 3.7% of their instructions in performing virtual function calls [1]. They go the additional step of saying that this overhead is likely to increase moderately on future processors. We believe that this will not be the case if compilers are enhanced to better utilize newer architectural features in order to reduce the additional time and/or instruction overhead. In this study, we present such a compiler optimization that applies predication on interleaved virtual function calls to eliminate unnecessary loading of virtual function tables (VFT) for objects whose type cannot be determined statically. By eliminating redundant loads , this technique will minimize the time spent accessing the memory system, and will reduce potential stalls in the instruction stream. We can show that the cost of applying this technique will be no more than a single cycle if the predication is not successful. However, we believe this cost is worth the potential savings when considering the nature of OO programs, and the savings that can be obtained in light of the growing performance gap between processor and memory systems.

1.1. The mechanics of virtual function calls

In order to understand how the overhead of virtual function calls can be reduced, the mechanics of such a call should be understood. Given in Figure 1(a) is an example of a class hierarchy using single inheritance, and its use. The object and VFT layouts for this example are shown in Figure 1(b), which are based on a standard layout as described by Ellis and Stroustrup [2]. According to Srinivasan and Sweeney [3], there are four steps required to perform a virtual function call when implementing a standard VFT layout. These are:

1. Load the VFT which contains the entry for the called function (i.e. access the VFT via the vptr).

2. Access from the VFT the function entry point to branch to (i.e. load the address of *A::foo* or *B::fee*).

3. Adjust the reference to the object through which the function is being called, to refer to the sub-object that contains the definition of the function which will be called (i.e. load *A-offset* and add to *obj1*, or load *B-offset* and add to *obj2*). This is known as a *late cast*.

4. Branch to the entry point of the function.

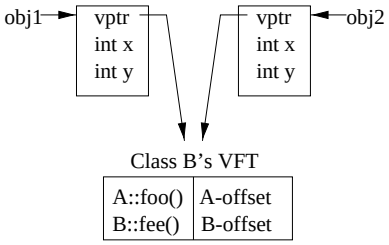
Class Declarations

```
class A {
  public virtual foo();
  public virtual fee();
  int x; };
class B : public A {
  public virtual fee();
  int y; };
```

Virtual Function Calls

```
A *obj1 = new B;
A *obj2 = new B;
...other code....
obj1->foo();
obj2->fee();
```

(a) Example class hierarchy and its use



(b) The object and VFT layouts

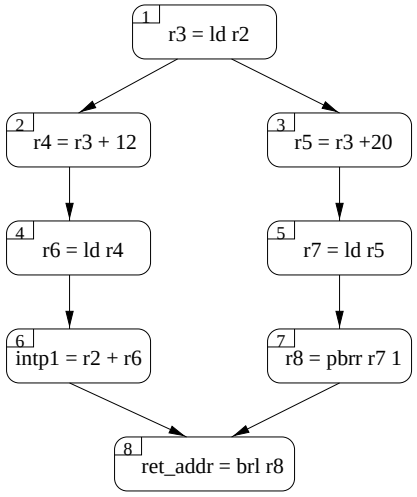
Figure 1. In C++ syntax, an example use of a single inheritance class hierarchy, and the resulting object and VFT layouts based on a standard implementation.

In a single inheritance hierarchy, an object has a single VFT in which all virtual function address are stored, thereby simplifying step 1 by accessing the VFT via the vptr. Whereas, in a multiple inheritance hierarchy, an object may have multiple VFTs with different virtual function addresses. Consequently, an extra step may be required known as an early cast. In an early cast, the reference to the object through which the function is being called is adjusted at runtime to refer to a sub-object whose VFT contains an entry for the called function. For either single or multiple inheritance, the late cast (i.e. step 3) is required only when the compiler stores offsets in the VFT rather than "thunk" code. The difference being that the "thunk" code handles the casting of the object to another object, and an offset provides a relative position of an object's definition within the object layout [3].

The overhead incurred by the above steps becomes apparent when translating the steps into the instructions that will be execute in order to perform the virtual function call. Given in Figure 2 is the low-level intermediate representation (LIR) and the data dependency graph for a virtual function call under the HPL PlayDoy (HPL-PD) architecture [4]. As can be seen from this Figure, the overhead of the virtual function call can be attributed to the load operations that are performed, and the dependencies between the load and the ALU operations (nodes 1,4 and 5). In most architectures, the memory loads will incur the largest overhead in comparison to the ALU operations, and will most likely stall the instruction stream if other non-dependent operations cannot be scheduled during the load.

```
r3 = ld r2          ;load the virtual function table (VFT)
r4 = r3 + 12        ;calc the address for the late cast offset
r5 = r3 +20         ;calc the address of the function entry point
r6 = ld r4          ;load the late cast offset
r7 = ld r5          ;load the function entry point
intp1 = r2 + r6     ;perform the late cast
r8 = pbrr r7 1      ;prepare to branch
ret_addr = brl r8   ;branch to the function entry point
```

(a) Low-Level Intermediate Representation



(b) Data Dependency Graph

Figure 2. The HPL-PD instructions for a virtual function call, and the data dependency graph.

In contrast to a static function call, as shown in Figure 3, the number of instructions to perform a virtual function call is greater, and consequently the performance is lower. A relationship can be established between the runtime perfor-

```
intp1 = r2                      ;set the this pointer
r3 = pbrr _$fn_foo__!DFv 1      ;prepare to branch to the function
ret_addr = brl r3               ;branch to the static function address
```

Figure 3. The HPL-PD instructions for a static function call.

mance of these function calls by expressing the runtime of the virtual function call (T_D) in terms of the runtime of the static function call (T_S). Ideally, T_D would be the same as T_S, however due to the extra instructions and the dependencies between these instructions, T_D will be larger by some delta. With this in mind, the relationship between T_D and T_s can be express as:

$$T_D = T_S + \triangle_t \quad (1)$$

The value of $\triangle_t$ is a factor of the extra instructions that are required for a virtual function call (IC_{Add}), the number

of cycles to issue an instruction (CPI), and the clock rate of the machine. Hence, $\triangle_t$ can be expressed as:

$$\triangle_t = IC_{Add} * CPI * \frac{1}{ClockRate} \quad (2)$$

and equation 1 can be rewritten as:

$$T_D = T_S + IC_{Add} * CPI * \frac{1}{ClockRate} \quad (3)$$

As for the CPI, this value can be expressed as the sum of the ideal CPI and the number of pipeline stall cycles per instruction. In the case of the virtual function call, the data dependencies between the load and ALU operations will be the cause of the pipeline stalls, and can be categorized as *Load Stall Cycles* and *ALU Stall Cycles*. For the purpose of this study we do not consider the stalls associated to the branch delay, and assume that the branch operation for a static function call will consume as many cycles as with a virtual function call. With this simplification of the CPI, we can express it as:

$$CPI = IdealCPI + \frac{Load\ Stall\ Cycles + ALU\ Stall\ Cycles}{IC_{Add}} \quad (4)$$

Since the clock rate of a machine is constant, the only way to reduce the runtime overhead of virtual function calls (T_D) is to reduce the number of additional instruction (IC_{Add}) and/or reduce the cycles per instruction (CPI). We propose a technique below that will do exactly this, by utilizing an HPL-PD feature known as predication to eliminate unnecessary loads of the VFT.

1.2. Applying predication to virtual function calls

A common practice within OO programs is to partition functionality into small, re-usable functions, also known as methods. Consequently, the average number of calls to methods in OO programs is higher than the number of calls to functions in procedural programs [5]. However, given that these methods are called through a finite set of objects, one would expect that a number of the methods are called with like objects (i.e instances of the same class). If this was the case, and the methods are virtual functions, then the calls to these functions will use the same VFT, as is shown in the example given in Figure 1(b). Therefore, if the compiler was able to interleave these calls, it could eliminate the redundant loads of the same VFT for each subsequent call after the initial call. This would reduce the number of load stall cycles (i.e. reduce the CPI) and reduce the pressure on the load/store units, which opens the opportunity to schedule other memory bound instructions. The compiler optimization presented in this study does exactly this by interleaving multiple virtual function calls for objects whose type cannot be determined statically, and applies predication to conditionally load the VFT when needed.

Predication supports conditional execution of individual operations based on boolean guards, which are implemented as predicated register values [6]. By use of predication, the compiler can transform a virtual function call so that certain operations (e.g. loading the VFT) are controlled by 1-bit predicate registers. The 1-bit predicate registers are read by the hardware during the instruction decode/register fetch cycle, and forwarded onto the execute cycle. Depending on the value of the qualifying predicate, the computed results of the guarded instructions are either applied towards the processor state, or discarded. As an example, Figures 4 and 5 show the predicated and non-predicated HPL-PD instructions, respectively, for two interleaved virtual function calls. These schedules are based on the example given in Figure 1(a), and on the assumptions given in Table 1. Note that the stalls shown in cycles *S1* and *S2* are due to structural hazards resulting from a single load/store unit with a latency that is dependent on the average memory access (AMA) time. Hence, the length of these stalls is variable, and in the case of the *S2* stall, it may be eliminated if the memory latency is short (e.g. a memory latency of 2 cycles).

Architecture	2-Way Issue VLIW
Integer ALU Units	2 units with a latency of 1 cycle
Load/Store Units	1 unit with a latency based on AMA time

Table 1. Assumptions applied to the schedules given in Figures 4 and 5

Cycle	VLIW Instruction		
1	r3 = ld r2		load obj1's VFT
S1	** Structural	hazard stall **	
2+S1	r10 = ld r9		load obj2's VFT
3+S1	r4 = r3 + 12	r5 = r3 + 20	calc. address for obj1's offset & function entry pt.
S2	** Structural	hazard stall **	
4+S1+S2	r6 = ld r4		load obj1's offset
5+S1+S2	r11 = r10 + 16	r12 = r10 + 24	calc. address of obj2's offset & function entry pt.
6+S1+S2	** Remaining	instructions **	

Figure 4. The schedule of non-predicated HPL-PD instructions for the two virtual function calls shown in Figure 1(a).

As shown in the schedule where predication is used, if the objects are of the same class type, then the redundant loading of the VFT is eliminated at cycle *2+S1*. This would avoid the stall cycles at *S2*, and provide an opportunity to schedule another memory bound instruction in its place (i.e. in the *S2* cycle). On the other hand, if the objects are of different class types, then all loads are performed and the length of the schedule is the same as when predication is not used.

In order to quantify the benefits of predication, we should first determine the cost of loading all VFTs, as would normally occur. In a schedule without the use of predication, the number of VLIW instructions required to load the VFTs

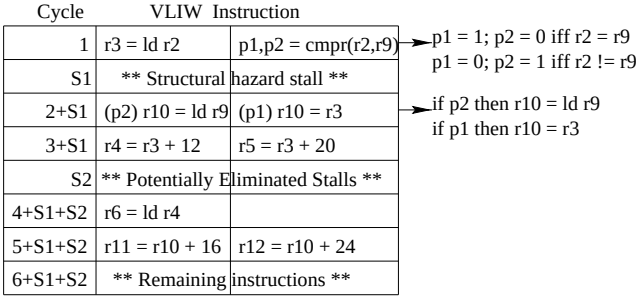


Figure 5. The schedule of the predicated HPL-PD instructions for the two virtual function calls shown in Figure 1(a).

is $\frac{V}{min(N,V)}$, where *V* is the number of interleaved virtual function calls, and *N* is the maximum number of load operations allowed per VLIW instruction. Each VLIW instruction will complete the loading of the VFTs in the time period for which it takes to locate the VFTs within the memory hierarchy (i.e. the average memory access cycles). This is a factor of the hit cycles, miss rate and miss penalty at each level in the memory hierarchy. For simplicity, we will refer to this as the AMA cycles, which is a based on the average memory access time as shown by Hennessy and Patterson [7]. Hence, the average number of cycles required to load the VFTs can be expressed as:

$$Avg.\ Load\ Cycles = \frac{V}{min(N,V)} * AMA\ cycles \quad (5)$$

and the AMA cycles for a 3-level memory hierarchy with an L1 and L2 cache and main memory, is:

$$AMA\ cycles = Hit\ Cycles_{L1} + Miss\ Rate_{L1} * (Hit\ Cycles_{L2} + Miss\ Rate_{L2} * Hit\ Cycles_{Main\ Memory})$$

By use of predication, the number of operations to load the VFTs is reduced by the cardinality of the set of virtual function calls (*S*) that use the same object type as the first virtual function call. In other words, once the first VFT is loaded, each subsequent function call using the same VFT can eliminate its load operation. Since the first virtual function call will load the VFT into the L1 cache, each eliminated load from the subsequent virtual function calls will reduce the overhead by the number of cycles required to access the L1 cache, assuming a nonblocking cache using a simple hit-under-one-miss scheme. We can express the number of cycles to load the VFTs with use of predication as an extension to Equation 5:

$$Avg.\ Load\ Cycles\ with\ Predication = \lceil \frac{V - |S|}{min(N,V)} \rceil * AMA\ cycles \quad (6)$$

Given in the schedule shown in Figure 5, where the number of interleaved virtual function calls is 2, and the assumed latencies and miss rates for the different levels in the memory hierarchy are as given in Table 2, the average load cycles with predication is:

$$Avg.\ Load\ Cycles\ with\ Predication = \lceil \frac{2 - |S|}{min(1,2)} \rceil * (2 + 0.2 * (7 + 0.12 * 35))$$

Level	Latency	Miss Rate
L1 Cache	2	20%
L2 Cache	7	12%
Main Memory	35	

Table 2. Example latencies and miss rates

In the case where the virtual functions are called with different types of objects, the set of virtual function calls (*S*) is empty. Whereas, if called with like objects then *S* is {obj2->fee}. The average load cycles required between these two cases is 8.48 and 4.24 respectively. Since in some cases the load cycles can be masked by overlapping non-dependent instructions with the loads, the reduction of load cycles by use of predication cannot be translated into a speedup for the virtual function call. However, when the load cycles cannot be fully masked, then the enhanced speedup resulting from the use of predication is approximately the ratio $\frac{Avg.\ Load\ Cycles\ without\ Predication}{Avg.\ Load\ Cycle\ with\ Predication}$. One would expect that as the the number of interleaved virtual function calls using like objects increases, so will the enhanced speedup.

What is missing from the average load cycles with predication is the cost of performing the predication (i.e. setting the predicate registers and evaluating the predicated instructions). Given an architecture that would allow a maximum of *N* loads in a single cycle, where *N > 1*, the use of predication would replace *N - 1* loads with compare-to-predicate operations (*CMPR*). Since the *CMPR* operation has a latency of 1 cycle, which is traditionally less than the latency of the replaced load operations, the impact of using predication thus far is a reduction in cycles. The *N - 1* replaced loads then become predicated loads, which can be scheduled in the cycle following the initial load. If any of these predicated loads are executed, then they would complete execution one cycle after the initial load, based on an average memory access time. Thus, the cost of using predicated loads when the object types do differ can be a single cycle. However, as shown in the schedule from Figure 5, the cost is 0 cycles when the load operation is not replaced but is merely predicated (i.e. N = 1), and the predicated load and assignment operations can be scheduled on the same cycle.

The other case to consider is when the objects are the same type, and the assignment of the VFT must be performed. This assignment cannot take place until the initial load has completed. In an architecture that allows *N* load operations in a single VLIW instruction, one can safely assume will also support *N* assignment operations in a single VLIW

instruction. Consequently, if all *N - 1* predicated assignment operations were to be executed, they could be scheduled in the same cycle and would complete in a single cycle (assuming a latency of 1 for ALU operations). Interestingly enough, if there exists a mix of virtual function calls that do and do not use the same object types, the predicated assignment operations can be scheduled on the last execution cycle of the predicated load operations since these operations are independent of each other. Hence, we can safely say that the cost of using predication in virtual function calls will cost one cycle in the worst case, thereby modifying Equation 6 to be:

$$Avg.LoadCycleswithPredication = \lceil \frac{V - |S|}{min(N, V)} \rceil * AMAcycles + 1 \quad (7)$$

In terms of Equation 3, predication reduces the number of load cycles which can potentially reduce the number of load stall cycles. The load stall cycles are used to compute the CPI, as shown in Equation 4. On the other hand, the instruction count which is also used to compute the CPI, increases due to the additional compare-to-predicate and assignment instructions. However, the predicated instructions (i.e. the VFT load and assignment) may or may not be committed to the processor state. This can impact the CPI. Thus, to accurately reflect the CPI on architectures that support predication, a *raw CPI* and a *useful CPI* should be considered. The raw CPI reflects all instructions, regardless of if they are predicated. Whereas, the useful CPI reflects only those predicated instructions that are committed. As for this study, we consider only the raw CPI which is an upper bound to the actual CPI. With this in mind, we can express T_D with predication as:

$$T_D\ with\ predication = T_S + [(IC_{Add} + 2) * (CPI - \frac{Eliminated\ Stall\ Cycles}{(IC_{Add} + 2)}) + 1] * \frac{1}{ClockRate}$$

2 Other techniques

This application of predication towards virtual function calls is not limited to single inheritance hierarchies, as our example depicts. In fact, this concept can be further extended with multiple inheritance hierarchies to reduce the cost of the early cast. For instance, if two virtual functions are contained in the same VFT, and the objects with which these functions are called are siblings in the class hierarchy (i.e. they inherit from the same base classes), then both the early cast and the loading of the VFT can be eliminated for one of the calls.

Another application of predication to reduce the overhead of virtual function calls is to use it in conjunction with runtime class tests, or also known as *I-Call If Conversion* [8]. This would convert the virtual function calls into static function calls, and reduce the branch mispredicts that are generally seen with this type of conversion.

3 Summary

To minimize the runtime overhead associated to virtual function calls, one must either reduce the number of operations required to perform a call, or reduce the CPI for the call. With a basic understanding of the mechanics of virtual function calls, one can see why they incur this overhead, and how this overhead might be reduced in light of newer architectural features. Predication is one such feature that could be used to conditionally eliminate loading the same VFT for multiple virtual function calls. By eliminating redundant load operations, we can minimize the time spent accessing the memory system and reduce potential stalls in the instruction stream. If the load operations cannot be reduced, then the cost of using predication, in the worst case, is a single cycle. We believe this cost is worth the potential savings when considering the nature of OO programs, and the savings that can be obtained in light of the growing performance gap between the processor and memory systems.

References

[1] K. Driesen and U. Holzle. The direct cost of virtual function calls in C++. In *Proceedings of the Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA '96)*, October 1996.

[2] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[3] H. Srinivasan and P. Sweeney. Evaluating virtual dispatch mechanisms for C++. Technical report, IBM Research Division, Jan 1996.

[4] V. Kathail, M. Schlansker, and B. Ramakrishna Rau. HPL PlayDoh architecture specification: Version 1.0. Technical report, Hewlett-Packard Computer Systems Laboratory, HPL-93-80, Feb 1994.

[5] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. In *Journal of Programming Languages, 2:4*, 1994.

[6] P. Y. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 2nd Edition, Morgan Kaufmann, San Francisco, CA, 1996.

[8] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *ACM Principles and Practices of Programming Languages, Portland Oregon*, 1994.

Hardware Support for Profiling Java Programs

Nathan M. Hanish and William Cohen
hanishn@itsc.uah.edu cohen@ece.uah.edu
Department of Electrical and Computer Engineering
University of Alabama in Huntsville
Huntsville, AL 35899

Abstract

Assuming the Java version of a program provides good performance, many programmers are interested in using Java as a replacement for many traditional programming languages because of the portability of Java and the extensive runtime libraries. However, in many cases the performance of the Java code requires improvement before it is acceptable. Profiling provides an effective means of identifying the sections of code that consume the most processing time and are the best candidates for optimization.

A prototype low-overhead, time-based profiling system has been developed for the Kaffe Java Virtual Machine's (JVM) Just-In-Time (JIT) i386 translator using the high-resolution timestamp register of the Intel Pentium processor. Experience with this approach suggests that a ''virtual time'' register would be a useful addition to the processor to simplify measuring the performance of multithreaded programs. Direct user control of the performance monitoring hardware would reduce the cost of measuring multiple performance metrics on a per-method basis.

1. Introduction

As a modern programming language whose programs can be executed on virtually any machine without recompilation, Java holds the power to revolutionize the manner in which applications are developed and delivered for industrial and research purposes. Unfortunately, Java is often considered a slow, inefficient language, but the performance of Java programs has been steadily improving and estimates have been made that Java can provide performance that is about 70% of traditional compiled languages [4]. However, there is still a need to obtain the best performance possible out of particular programs executing on the Java Virtual Machine (JVM). This type of effort requires the use of runtime profiling information. This profiling information can either be used by the programmer to change sections of the code to improve performance or it can be used by the runtime system to efficiently apply optimizations to sections of the code as proposed by Hölzle [3] and implemented in HotSpot [6]. This profiling information is essential for quickly focusing the effort on sections of code that most impact the overall performance of the program.

The most commonly used metric is time. If a section of the program takes very little time to execute, further optimization of that section will have little effect on the overall performance of the program. Traditionally, expensive function calls were used to obtain time data from off-chip sources, causing the measurement to perturb the data being collected. Many processors, including the Intel Pentium [5], Compaq Alpha [10], and PowerPC [8], include timestamp registers, which can be read with a couple of simple instructions and provide time resolution on the order of tens of nanoseconds. This processor hardware reduces the cost of measuring the amount of time required to execute a section of code. The abilities of the Intel Pentium's timestamp hardware were exploited on a prototype profiling system incorporated into the Kaffe Java Virtual Machine's (JVM) [12] Just-In-Time (JIT) translation system. The profiling minimized the cost of obtaining timing information by inserting in-line machine language instructions to read the processor's timestamp register.

Given the experiences of incorporating hardware-assisted profiling into a JVM, some observations were made to improve the hardware support for monitoring the performance of the Java runtime system. Java is multithreaded and an additional timestamp register that contains the virtual time for the thread would be useful. The wall-clock time does not indicate the amount of time that the thread is actively using the processor, software schemes that track the time that the thread spent idle must be implemented [7]. In addition to a high resolution timer, most processors provide performance monitoring hardware that can measure other aspects of the processor performance, such as the number of cache misses, memory accesses, and instructions executed. Although the performance monitoring hardware allows the values to be directly read by application programs, setting the configuration of performance monitoring hardware is limited to the kernel. Direct control of the performance monitoring hardware by application programs would avoid expensive kernel calls to change the metrics being

collected, enabling finer grained data collection, e.g. changing the metrics being collected on each method invocation.

Section 2 describes the approach used to provide profiling in the Kaffe JVM. The environment used for the experiments and the results of the instrumentation are in section 3. Finally, section 4 summarizes the work.

2. Instrumentation Architecture

The instrumentation was designed to provide an automated collection of high-resolution time measurements on each method in the executing program. The use of a very high-resolution clock (period same as processor clock) allows timing information to be gathered on methods that run for very short amounts of time. The high-resolution measurements also allow meaningful data to be collected without resorting to repeated calls to the same method to increase the amount of time spent in the method. Having repeated calls to a method may perturb the measurements because of garbage collection and other interfering threads running concurrently. The instrumentation process is automatic, occurring when the bytecodes are translated into native machine language, so the instrumentation collects data on each bytecode method executed regardless of whether or not the user has Java source code for it.

Several modifications were made to the Kaffe JVM to provide time-based profiling. A option on the command line, ‘‘-prof’’ activates the instrumentation. Fields added to the structure that describes each method store the profiling information. The revised JIT code generator produces native code versions of the methods. Finally, the exit code of the JVM was modified to print the number of times each method was called, the amount of time spent in each method, and the amount of time spent in the method and its children.

The timestamp register on the Pentium processor counts the number of cycles since the processor was reset. The application code can read this register with a single instruction, RDTSC, which stores the 64-bit value in two 32-bit registers, EAX and EDX. There are very few registers in the IA-32 architecture. Thus, EAX and EDX are used as general purpose registers by the JVM and in most cases the values in EAX and EDX need to be saved before the timestamp is read and restored after the time is recorded.

The instrumentation stores the information on a per-method basis. Each method loaded into the JVM has a structure that describes it. Fields added to the structure accumulate the number of times the method is called, the total time spent in the method and its children, and the time spent in its children. The fields are zeroed when the method is translated from Java bytecode to native machine language.

The JIT system generates in-line machine code to profile each method. In the prologue of the method, the invocation count for the method is incremented and the current time stamp is subtracted from the method’s total time. In the epilogue of the method, the current time stamp is added to the method’s total time, yielding the net time spent in the method and the children.

To determine the time spent in the children methods, each method invocation in the generated code is preceded and followed by time measurements. The time measurement preceding the method invocation is subtracted from the time spent in the children and the time measurement following the method invocation is added to the time spent in the children. These operations add the net time spent in the child to the total spent in the children methods.

This approach samples the time stamp register twice as many times as theoretically required. The time is sampled just before leaving a method and just after entering a method. However, the JVM runtime system does not guarantee that the leaving of one method is immediately followed by the entry into the next because of the trampolines used to start the JIT translation. Making the assumption that the two times are the same would incorrectly attribute time for JIT translation to one of the methods. The cost of sampling the time stamp register is relatively low. By keeping the profiling information within the boundaries of the method, the addresses of the structure storing the profiling information is known when the JIT is performed. If the profiling data crosses method boundaries, additional code is required to dynamically determine the appropriate structure to update because the same method invocation may invoke different code depending on the object. Thus, the selected approach samples the time more often than absolutely needed, but it yields more accurate estimates of the time spent in each method and still provides good performance.

3. Results

Several experiments were performed on a 100MHz Pentium running Linux RedHat 4.2 and the modified version of Kaffe. The profiling modification performed reasonably well and its reported times agreed with the times reported by the currentTimeMillis used in the Symantec benchmark suite for the various parts of the benchmark. Table 1 summarizes these measurements and shows the difference between the measurements for the same run of the benchmark. Another aspect of the instrumentation is the amount of time that is measured by the profiling compared to the total time taken to execute the program being measured. Subtracting the time spent in the method’s children from the total time recorded for the method provides the time spent in the method, the selftime. The selftimes for all the methods were summed to determine the amount of time actually observed by the

Table 1: Comparison Two Method of Measuring Benchmark Runtime.

Benchmark	current Time Millis (ms)	IA-32 Time Stamp (ms)	Percent Diff.
Sieve	3733	3737.957	0.03317%
Array	13745	13775.978	0.05628%
Dhrystone	14328	14355.194	0.04740%
Fibonacchi	16828	16867.035	0.05792%
BidirectionalBubble SortAlgorithm	23103	23154.048	0.05518%
BubbleSortAlgorithm	26876	26938.705	0.05826%
Hanoi	35885	35970.236	0.05931%
Tree	87766	87977.295	0.06011%

instrumentation. Table 2 shows the usertime measured by the Unix time command for the Symantec [11], Linpack [2], jBYTEmark v0.9 [1] benchmarks. Additional measurements were made with Pizza [9], a compiler that provides a superset of Java, compiling the other benchmarks. The ratio of usertime versus sum of the times spent in each method as measured by the JIT instrumentation is recorded in the last column of Table 2. For the simple benchmarks no more than 60% of the time was spent in the bytecode. Much of the time was spent in either native methods or the JVM support code. Existing C profiling tools can be used to determine the time spent in the JVM code. The current version of the profiling also instruments the Java Native Interface (JNI) wrapper providing an estimate for the amount of time spent in C functions called from Java.

The profiling added about 2% to the overall runtime. This overhead is expect to increase when improvements are made to the quality of the code generated by the JIT

Table 2: Comparison of coverage of JVM runtime by profiler.

Benchmark	User time utime(s)	Total self time (s)	self / utime
Symantec	264.3	101.3	0.383
Linpack	1.56	0.510	0.327
jBYTEmark	75.4	44.0	0.583
pizza (Symantec)	12.9	0.619	0.048
pizza (Linpack)	11.6	0.435	0.038
pizza (jBYTEmark)	20.8	2.31	0.111
HelloWorldApp	0.92	0.0042	0.0046

translator and improvements are made to reduce the amount of time spend in the JVM runtime system.

Table 3 shows the output of the profiler for a simple HelloWorldApp class, which simply prints ‘‘Hello World!’’ The main method in the HelloWorldApp and everything it calls requires relatively little time to execute, 66.0118ms, and 66.0071ms of that time is spent other methods called from that method. Significantly more time is spent in the class initializer java/lang/System.<clinit>(), 390.254ms.

Measurements for the Volano benchmark [13] shown in Table 4 illustrate the weaknesses of the current instrumentation in measuring programs that use multithreading and exceptions. The measurements were sorted based on the total times and the largest total times are listed in Table 2. The Volano benchmark required 284 seconds on a uniprocessor machine. The instrumentation makes no distinction between a thread doing useful work and being idle. Thus, the total time spent in COM/volano/cb.run()V and COM/volano/x.run()V are 200 times larger than the time actually spent by the processor in these methods. Another oddity observed in Table 4 is the negative time spent in the children for COM/volano/x.run()V. This is due to Volano using an

Table 3: Top 20 times for HelloWorldApp.

count	total (ms)	children (ms)	method-name
1	506.53	506.375	java/lang/ThreadGroup.add(Ljava/lang/Thread;)V
1	390.254	281.022	java/lang/System.<clinit>()V
3	194.99	194.98	java/lang/Class.forName(Ljava/lang/String;)Ljava/lang/Class;
3	194.546	0	java/lang/Class.forName0(Ljava/lang/String;)Ljava/lang/Class;
1	129.966	129.878	java/util/GregorianCalendar.<clinit>()V
1	129.057	129.052	java/util/GregorianCalendar.<init>(III)V
1	116.842	116.838	java/util/Calendar.<init>()V
1	101.821	101.818	java/io/FileInputStream.<clinit>()V
3	93.5362	93.5265	java/lang/System.loadLibrary(Ljava/lang/String;)V
1	91.3835	90.0128	java/util/Locale.<clinit>()V
22	89.1518	89.1078	java/util/Locale.<init>(Ljava/lang/String;Ljava/lang/String;)V
22	86.9751	86.9022	java/util/Locale.<init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V
22	79.5602	79.5254	java/lang/String.toLowerCase()Ljava/lang/String;
22	76.6465	75.6919	java/lang/String.toLowerCase(Ljava/util/Locale;)Ljava/lang/String;
1	69.804	69.6825	java/lang/Runtime.<clinit>()V
44	67.4528	67.3652	java/lang/Character.toLowerCase(C)C
74	67.1331	15.2901	java/lang/Character.getCharProp(C)Ljava/lang/Character$CharacterProperties;
1	66.0118	66.0071	HelloWorld.main([Ljava/lang/String;)V
1	65.2982	65.2925	java/io/PrintStream.println(Ljava/lang/String;)V
1	59.0576	59.0528	java/io/PrintStream.print(Ljava/lang/String;)V

Table 4: Volano measurements.

count	total (ms)	children (ms)	method-name
400	1.18762e+08	1.18762e+08	java/lang/Thread.run()V
200	5.94153e+07	5.94153e+07	COM/volano/cb.run()V
200	5.93471e+07	-4.15756e+11	COM/volano/x.run()V
2799	5.87699e+07	5.87697e+07	java/lang/Object.wait()V
2799	5.87697e+07	5.87696e+07	java/lang/Object.wait(J)V
2799	5.87695e+07	5.87689e+07	java/lang/Thread.waitOn(Ljava/lang/Object;J)V
2799	5.87688e+07	0	java/lang/Object.wait0(J)V

exception to exit a child method. Thus, one of the times for returning from a child is never added to the children time, yielding a negative number.

Changes to the performance monitoring hardware could simplify the data collection process and improve the quality of the measurements. There are really two types of time measurements of interest for profiling: wall-clock time and virtual time. The processor only directly supports the measurement of wall clock time via the timestamp register. The problem becomes apparent when trying to measure multithreaded programs such as the Volano [13]. The wall-clock time does not give an accurate measurement of the time a method spends executing in a multithreaded program because the method may be suspended for significant amounts of time and during that time it does not execute. An additional virtual timestamp register implemented in hardware would be a useful. This would be a user readable and writable register that would increment at the same rate as the regular timestamp register. When a context switch occurs, the value in the virtual timestamp register would be reset to that thread's virtual timestamp value and continue counting where the thread left off. This would provide an accurate measure of the amount of processor time that various methods consume. Software that virtualizes the IA-32 timestamp register has been developed, but it requires 90 cycles per measurement [7]. A trivial amount of hardware could reduce the cost of this operation by an order of magnitude.

The IA-32 architecture has very few registers to store values and the instruction that reads the time stamp store the values in registers that are used for many other purposes. This results in additional instructions being required to save the values before the timestamp operation and to restore the registers after the time stamp operation.

There is little reason to make the control of the performance monitoring hardware privileged. Having a performance monitoring hardware that is directly under user control would eliminate the costly kernel calls required to switch the instrumentation from user code. With direct user control it would be possible to switch the performance metrics being monitored within the JVM or even from method to method without incurring too much overhead. If the kernel really requires dedicated performance monitoring, the cost of performance monitoring hardware is relatively small and a duplicate unit could be created that is only accessible from kernel-mode.

4. Conclusion

For wide scale adoption of Java, tools that allow programmers to identify areas of code that require improve are essential. The JIT modifications to the Kaffe JVM provided an adequate means of profiling traditional sequential benchmarks, and was able to indicate which methods consume the most time in the Java program. The performance monitoring hardware provided by the processor was crucial to the efficient implementation of the profiling. The development of the profiling system exposed some of the limitations of the current hardware support for performance monitoring.

Relatively simple changes in the performance monitoring hardware will improve the quality of the data collected. Having a separate virtual timestamp register will allow more efficient profiling of multithreaded programs. Multithreaded programs will become much more common due to the Java's built in support of threads and the interest in networked programs. Allowing direct user access to the performance monitoring hardware will enable the JVM to perform more sophisticated performance monitoring without resorting to costly kernel calls to change the metrics being monitored.

Acknowledgments

We thank Edouard G. Parmelan for refining our prototype instrumentation and incorporating it into the Kaffe JVM.

Bibliography

[1] ''Benchmarking Java.'' Byte, May 1998.

[2] J. Dongarra and R. Wade. ''Linpack,'' 1998. http://www.netlib.org/benchmark/linpackjava/.

[3] Urs Hölzle, *Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming*, Ph.D. thesis, Computer Science Department, Stanford University, August 1994.

[4] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen-mei W. Hwu, ‘‘Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results,’’ *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[5] Intel, *Architecture Optimization Manual*, Mt. Prospect, Illinois,1997, Intel, http://www.intel.com/design/pentium/manuals/24281-6.htm, order number 242816-003.

[6] Sun, *THE JAVA HOTSPOT™ PERFORMANCE ENGINE ARCHITECTURE*, http://www.javasoft.com/products/hotspot/whitepaper.html, April 1999.

[7] Charles E. Leiserson, personal communication, August 1999.

[8] Motorola, PowerPC 604/604e User’s Manual, MPC604EUM/AD, Motorola, March 1998.

[9] Martin Odersky and Philip Wadler, ‘‘Pizza into Java: Translating theory into practice,’’ *Proc. 24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[10] R. L. Sites and R. T. Witek. Alpha AXP Architecture Reference Manual, Digital Press, Newton, Massachusett, second edition, 1995.

[11] Symantec’s Benchmark Test Suite, http://www.symantec.com/jit/jittests.zip 1999.

[12] Transvirtual, *Kaffe*, http://www.transvirtual.com/kaffe.html, May 1999.

[13] ‘‘Java Benchmarks: Volanomark,’’ http://www.volano.com/mark.html, July 1999.

Session 4

Microarchitectures for Java

VLSI Architecture Using Lightweight Threads (VAULT) - Choosing the Instruction Set Architecture

I Watson, G Wright, A El-Mahdy
University of Manchester, UK.
watson@cs.man.ac.uk, gwright@cs.man.ac.uk, aelmahdy@cs.man.ac.uk

Abstract

The VAULT project is concerned with the design of a 'multi-processor on a chip' aimed specifically at multi-threaded Java implementation. It has wide ranging aims that require research in a variety of hardware and software areas. The project is still in its early stages and most of the work is still to do. This paper provides an overview of the project as envisaged currently and then examines some of the initial work in detail. In order to perform a comprehensive evaluation of the VAULT approach, it was thought necessary to perform detailed instruction level simulation. A single CPU structure therefore needed to be defined to form the basic building block of the system and hence the simulator. One of the first decisions needed was the basic Instruction Set Architecture of the CPU. The reasons behind the choice are examined using results obtained by detailed instrumentation of the Java Virtual Machine and the running of a variety of Java benchmarks.

1.Introduction

The VAULT project has two distinct but closely connected aims: firstly to explore the potential of future VLSI to produce a 'multi-processor on a chip' and secondly to provide support for high performance Java based systems of the future.

The combination of these two issues is neither accidental nor arbitrary. The case for considering single chip multi-threaded and/or multi-processor structures to utilize future VLSI technology is well known. However, most proposals in this area have either been extensions of current super-scalar designs or integrated versions of conventional multi-processors [1]. Concentration on the Java environment allows us to consider new approaches which can optimize the software performance, while at the same time benefit from the natural multi-threaded structure, to produce high performance, low power hardware.

The environment is significantly different from that required to support serial or parallel C or FORTRAN based code for which current processors are optimized. The most obvious differences are:-

- Execution by dynamic compilation of bytecode.
- Object oriented code with heavy use of method calling, dynamic binding, object access (via indirections) and garbage collection.
- Dynamic linking and loading of objects.
- Program level multi-threading.
- Particular importance of multi-media applications.

It is believed that hardware structures can be tailored to these needs with a resulting performance increase. It is also important to note that many such systems will need to be portable and hence power dissipation is a significant issue.

It is assumed that compatibility with previous hardware is not necessary (this is probably essential to permit the investigation of novel 'on-chip' mechanisms). Dynamic binary translation can be used to run 'legacy' software if necessary.

This paper outlines the overall features of the VAULT architecture, which are aimed at realizing high performance for the Java environment. However, the project is at an early stage and much of the detail is still being explored. The major detail of the work described here is concerned with the selection of an Instruction Set Architecture (ISA). A selection of programs from the JavaSPEC [2] benchmarks has been used to analyse the various ways in which bytecode can be executed and the resulting overheads which occur. This analysis suggests that a register windows based CPU would provide optimum performance. Assuming that this approach is followed, the final section describes a more detailed analysis of the benchmark execution in order to ascertain the numbers of registers and windows necessary to achieve that performance.

2.Project Principles

The following are the principles and features which are guiding our research. This is currently in an early stage so that the detail of some of these issues has not yet been explored.

1. Parallelism through multiple simple CPUs rather than exploiting ILP etc.

- Natural exploitation of threaded parallelism - potentially higher performance than the alternative of time sliced execution on super-scalar CPU as thread count increases.
- Complexity/performance ratio per CPU lower leading to more efficient silicon use.
- Power/performance ratio per CPU lower - particularly important for portable computing.
- Simple processors imply significantly reduced design effort.

2. Processor structure optimized for support of (dynamically) compiled Java and multiple thread support.

   - Register windows structure for efficient support of frequent subroutine (method) calls.
   - Multiple register set 'heap' with hardware assisted allocation and spilling to allow frequent thread switching within a CPU.
   - Caching structure tailored to object accessing and dynamic binding.

3. Inter processor communication at register and cache level via special bus structures.

   - On-chip communication will enable specialized communication paths to be implemented at high speed.
   - Support for very lightweight thread creation. Use of dynamic thread creation. Use for 'real' parallelism (loops etc.) within Java level threads for higher parallel performance (e.g. for multi-media computations).
   - Support for dynamic load balancing. Rapid exchange of load information and rapid task creation minimize 'feedback system instability'.
   - Ability to query remote caches a significant aid to task placement.

4. Processor support for multi-media applications.

   - Multi-dimensional cache structures.
   - Multi-media processor functions?

5. Dynamic compilation (for parallelism).

   - Dynamic compilation for optimal use of single processor structure (Hot Spot etc.)[3].
   - Decisions on size of parallel tasks best left until runtime.
   - Decisions about how to divide (e.g. data sets) best left until run time.
   - Data representations can be altered dynamically? (e.g. tree structured arrays)

There have been a number of architectures proposed for the efficient execution of object-oriented languages, most of them aimed at Smalltalk. The SOAR [4] processor had a register windows structure but these were not organized as a freely allocated heap like VAULT. Most of the other SOAR features such as tagged data are not applicable in a Java environment. The Mushroom [5] project examined novel memory mechanisms for object accessing and caching which may be relevant to VAULT.

We have (very) recently become aware of some of the details of the Sun MAJC architecture [6]. This appears to share many of its higher level aims with VAULT. The major differences appear to be at the individual processor level where they propose the use of multiple function units and a VLIW ISA.

3.Choosing the VAULT CPU ISA

It is tempting to think that the correct way to approach the design of hardware to execute Java efficiently is to produce a stack based CPU. This might either implement the full functionality of the Java Virtual Machine in hardware or at least provide a simple stack based ISA into which the bytecodes can readily be translated.

It was felt that the first of these options was against basic RISC philosophy and was thus unlikely to lead to optimum performance. The picoJava [7] project has already investigated the second approach. It uses stack caching techniques coupled with hardware supported instruction folding to overcome some of the inherent disadvantages of a stack based ISA concerned with the movement and duplication of operands. Such a processor is particularly suited to embedded applications where it requires minimal software support. A number of other projects have also studied similar hardware techniques.[8][9]

We were not convinced that, if one assumed the use of sophisticated JIT or dynamic compilation techniques, a stack based structure would outperform a more conventional register based ISA. However, due to the heavy use of method calling in many Java applications, we thought that the use of register windows might be beneficial. Although it is, of course, possible to study Java implementations on real processors which exhibit the various design alternatives we felt that there was a need to perform an evaluation using a common methodology. We therefore studied a number of different ISAs together with appropriate software support using a variety of Java benchmarks.

In order to produce a useful comparison, it was necessary to postulate some simple cases which represented distinct points in the design space. We chose to compare the following models of execution:-

- **Bytecode** - A direct execution of Java bytecode assuming no optimization.
- **Folding_2** - Java bytecode execution assuming the folding optimizations used in the picoJava-1 processor (up to two instructions are folded).
- **Folding_3** - Extends the Folding_2 model to consider 3 instruction folding
- **Folding_4** - Java bytecode execution assuming the folding optimizations used in the picoJava-2 processor (up to four instructions are folded).
- **Reg** - A simple register based ISA together with a 'state of the art' register allocation algorithm. We chose to base this on the techniques described for the Cacao compiler [10] as these seemed to represent an efficient but simple mechanism.
- **Regwin** - A register windows based ISA again using the Cacao algorithm.

We have implemented these models and added them to the Sun JDK JavaVM through which the JavaSPEC benchmark programs are analyzed. (The _227_mtrt program was omitted as it is a multithreaded program and we are concerned here with simple single thread characteristics.)

The nature of the benchmarks is described briefly in Table 1.

Table 1: Benchmark Programs

Program	Description
_201_compress	Lempel-Ziv compression
_202_jess	Java Expert Shell System
_209_db	Database functions
_213_javac	JDK 1.02 Compiler
_222_mpegaudio	Decompress MPEG-3 audio
_228_jack	Java version of yacc

The results are summarized in Figures 1 and 2. Figure 1 shows the dynamic bytecode execution frequencies for various bytecode classes; constant loads (const), local variables load/store (local), array load/store (array), stack operations (stack), arithmetic/logic (ALU), conditional/unconditional branches (branch), field load/store (field), method invocation (invoke) and other (ow). The results assume the Bytecode execution model. The high occurrence of const, local and stack operations, contributing around 50% of the total instruction count, is responsible for the stack overhead.

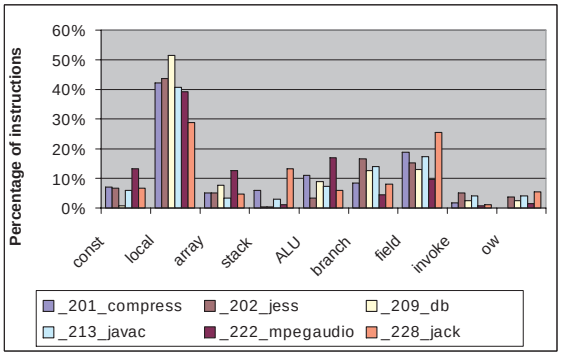


Figure 1. Dynamic instruction mix.

Figure 2 shows the number of instructions executed in each benchmark program for the six execution models. Folding_2 reduced the number of instructions by a maximum of 17% (for _209_db) and a minimum of 6% (for _202_jess); the average is 12%. Folding_3 and Folding_4 contribute another 3% at most with average of 1%.

Reg worked better than the folding models for four of the programs; reductions ranged from 26% to 44%. For the other two programs (_202_jess, _213_javac) there is a smaller reduction of 14%. As can be seen from Figure 1, these programs have nearly double (2, and 1.7 respectively) the number of method calls of the other programs, increasing the relative call overhead.

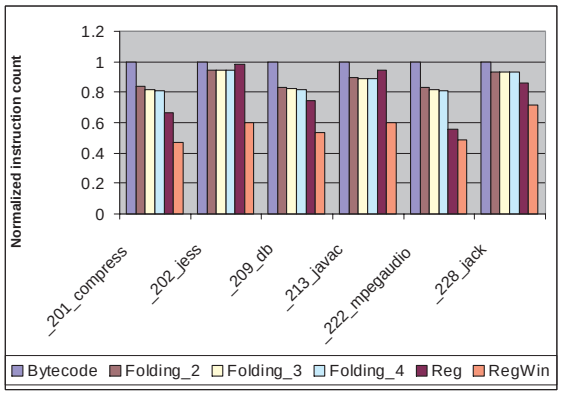


Figure 2. Relative instruction counts

Regwin was able to outperform all other models. This was expected observing, from Figure 1, that method calls account between 1% and 5% of the executed instructions. Regwin reduced the number of instructions by at least 40% for five programs. The remaining program (_228_jack) showed only a 29% reduction. This is attributed to the rela-

tively frequent stack (at least 8%) and infrequent local and ALU operations. This suggests less sharing of local values and hence the benefit of using registers is decreased.

We have not considered the effects of 'in-lining' in our study. This would undoubtedly narrow the gap between the Reg and Regwin results for some applications. However, there are limits to which in-lining techniques can be used for deeply nested programs and it is inapplicable in the general cases of both recursion and dynamic binding. It can be argued that most of the JavaSPEC benchmarks are not representative of programs written using the full Object Oriented style where the above issues will be of increased relevance. We therefore believe that achieving optimum performance on real method calls is important and hence our approach is justified.

4.Register and Window Usage

The previous analysis assumed an unlimited supply of registers and register windows. We re-instrumented the Sun JDK JavaVM to count the number of local variables accessed. Figure 3 shows the accumulated percentage of local variable usage assuming that all local variables are mapped into registers.

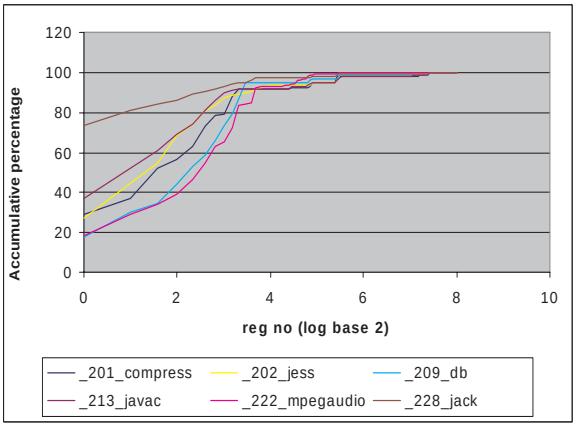


Figure 3. Cumulative percentage of register usage

The x-axis shows the number of registers required on a base 2 logarithmic scale (i.e., number of bits required to encode a local variable). The most register hungry application (_222_mpegaudio) has a 'knee' at 13 registers covering 91% of variable accesses. The next significant 'knee' is at 30 registers where several applications achieve 95% usage. This indicates that the decision is between 16 and 32 registers although a more detailed study of dynamic register usage in the presence of dynamic register allocation algorithms is required before a final decision is reached.

In order to determine the number of register windows, it is necessary to study the method call depth. We extended our instrumentation to provide this information. Figure 4 shows the call depth distribution for the various benchmark programs. The x-axis is the actual call depth, while the y-axis is the percentage of total instructions which get executed at that depth. The absolute value of the call depth is of minor importance, in fact the offset of 13 in Figure 4 is due to a set of 'wrapper' methods around the benchmark suite which are executed initially. These will, of course, require register window allocation and register 'spills' if the window total is exceeded but this will only occur once. The important characteristic is the width of the profile. Programs such as _209_db with a very narrow profile indicate that execution takes place with very shallow nesting while a broad profile like _202_jess indicates deep nesting.

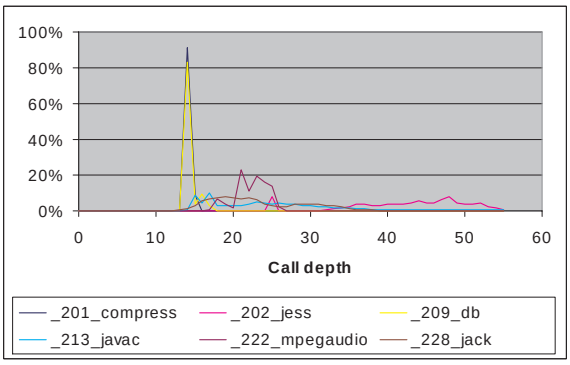


Figure 4. Call depth distribution

Flynn [11] suggests that a useful measure of the call depth of programs is the relative call depth defined as the average of absolute differences from the average call depth. Table 2 shows the relative call depth for the benchmarks used.This clearly distinguishes the different characteristics which are apparent from the graphical profile. However, it does not give an accurate figure for the actual number of windows needed.

Table 2. Relative Call Depth

Benchmark	201	202	209	213	222	228
Relative Call Depth	0.16	5.88	0.54	7.40	1.70	5.67

A more accurate estimate of the register window requirements is necessary before design decisions can be made. Our instrumentation was yet further extended to study the way in which the provision of windows affected the execution.

We simulated the benchmarks with varying numbers of

windows and measured the miss ratio, defined as the ratio of the number of window accesses resulting in window overflow or underflow to the total number of windows accesses. A window access takes place twice per method call, once on entry and once on exit. The results are shown in Figure 5.

As expected from the relative call depth figures, two of the benchmarks have a requirement for only a small number of windows and, for them, two might suffice. However, to achieve a low miss ratio (less than 0.02) for all benchmarks, eight register windows appear to be necessary. To emphasize, at this level over 98% of all method calls would not require register spilling.

From these experiments, it is believed that we have determined that a configuration of eight register windows each containing 16 or 32 visible registers would be sufficient to achieve good performance.

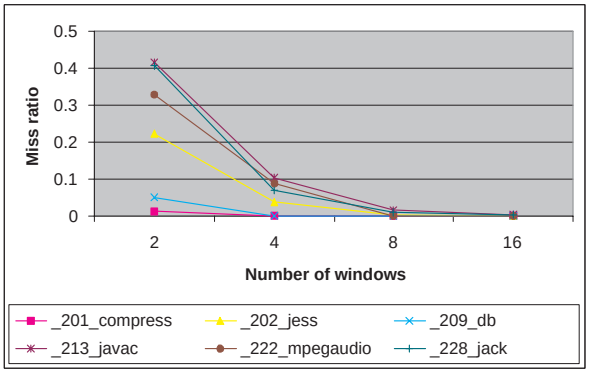


Figure 5. Window miss ratios

It is thought that the this configuration is of acceptable complexity for the processor design being considered.

5.Conclusions

This paper presents an overview of the VAULT project followed by the detail of the choice of an ISA.

The design issues are outlined together with possible solutions currently being investigated. The most important features of VAULT are thought to be:-

- Simple CPU structure with optimized support for Java like languages.
- On-chip multi-processor structure with fast thread synchronization facilities.
- Support for multi-media processing.

Initial results concerning the ISA design are presented which have demonstrated that an ISA using register windows results in a dramatic reduction of the stack overhead, far better than can be achieved by folding techniques. The register windows structure is important for reducing method call overheads.

We have verified by detailed simulation that the number of registers and register windows required to achieve good performance is modest and therefore consistent with the level of hardware complexity envisaged.

The work done so far has enabled us to narrow the design space to a level where we are able to embark on the construction of an instruction level simulator for a VAULT CPU. This work, together with compilation routes from Java bytecode (and C) is almost complete. Using this, we will be able to perform a very accurate verification of the design decisions presented above and make any adjustments to the CPU structure which are necessary. The next stage is to extend the simulation to the multi-processor structure in order that we can start to study the full potential of the VAULT approach.

6.References

[1] Doug Burger and James R. Goodman. Billion-transistor Architectures. *IEEE Computer*, 30(9):46-49, September 1997.

[2] SPEC JVM98 Benchmarks, Standard Performance Evaluation Corporation. http://www.spec.org/osg/jvm98.

[3].David Griswold. The Java HotSpot Virtual Machine Architecture. *White paper, Sun Microsystems*, March 1998. http://java.sun.com/products/hotspot/whitepaper.html.

[4] David M. Ungar. The Design and Evaluation of a High Performance Smalltalk System. *ACM Distinguished Dissertation.* MIT Press, Cambridge, Massachusetts, 1987.

[5] I.W. Williams, M.I. Wolczko and T.P. Hopkins. Dynamic Grouping in an Object Oriented Virtual Memory Hierarchy. *Proceedings ECOOP 1987* pages 87-96.

[6] Introduction to the MAJC Architecture. *Sun Microsystems*, August 1999. http://www.sun.com/microelectronics/MAJC/documentation/majcintro.html

[7] Harlan McGhan and Mike O'Connor. PicoJava: A Direct Execution Engine for Java Bytecode. *IEEE Computer*, 30(9):79-85, September 1997.

[8] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. Object-Oriented Architectural Support for a Java Processor. In E. Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 330-354. Springer, July 1998.

[9] Patriot Scientific Corporation. *PSC1000 Microprocessor.* http://www.ptsc.com/psc1000/index.html.

[10] Andreas Krall. Efficient JavaVM Just-In-Time Compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, October 1998.

[11] Michael J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett, Boston, 1995.

A Two Step Approach in the Development of a Java Silicon Machine (JSM) for Small Embedded Systems

H. Ploog · R. Kraudelt · N. Bannow · T. Rachui · F. Golatowski · D. Timmermann
Department of Electrical Engineering and Information Technology
University of Rostock, Germany
E-mail : hp@e-technik.uni-rostock.de

Abstract

In current solutions a Java Virtual Machine executes Java byte code by interpretation or dynamic compilation. To increase the execution performance we propose our experiences in the development of a processor architecture that can directly execute JavaCard 2.0 compliant byte code.

1. Introduction

JAVA has been developed for desktop and internet based systems but there are several implementations in the embedded area where specific Java advantages can be reused, so the idea behind JAVA and its benefits is successively transported into the embedded system-industry. In this paper we focus on *static embedded systems* in which is no need for dynamic class loading during runtime (e.g. car radio, cola machine, smart card). But the user of such systems also has the possibility to choose from a set of different applications.

To execute Java byte code on a platform there are at least 3 possibilities which are different in terms of execution speed:

- Interpreted execution
  The byte code is interpreted by software.
- Compiled execution
  In an additional step the Java byte code is compiled to a specific processor architecture so that there is no runtime overhead for inter-pretation. It is also possible to compile the byte code just in time (JIT), the so called dynamic compilation-technique.
- Direct execution (JSM)
  Java byte code can directly be executed by a real Java processor.

By real Java processor we consider a processor not only optimized for executing JAVA-code but capable of executing Java-code without a software implemented Java Virtual Machine.

In the last few years the importance of SUN's Java-technology raised up and it becomes a topic on many university and commercial research programs but only a small number of projects dealing with the development of a Java Silicon Machine (JSM) are known.

Many of the existing chips [7],[8] remap the Java byte code to a new (reduced) instruction set to speed up the performance. Glossner et al. described a system which is based on the same idea but they also used a multithreading architecture [3]. Another theoretical description of a Java processor architecture can be found in [6]. At the University of Zurich an ongoing project is called JAMA [4]. It seems that this processor will be designed for direct execution of JAVA byte code.

To our knowledge by now only three existing (real) Java processors are known: picoJAVA-I [12], microJava (picoJAVA-II) and JEM-1[13] . The latter one is designed by Rockwell Collins Inc. and the former one by SUN itself. Since microJava is based on the intellectual property module PicoJava-II one anticipates an increasing number of custom Java processors.

In this paper we describe our experiences in the development of a JSM for small embedded systems like smart cards.

The project is split into two parts. The first part is a software-based Java Virtual Machine suitable for 8051-processor like systems for architecture exploration, and part two is the JSM itself. The second part is still under development since this is currently a work in progress.

2. Differences between Java and Java for smart cards

A smart card is a single-chip computer based on an 8-bit microcontroller. The two most commonly used chips are Motorola's 6805 and Intel's 8051. These systems

contain three different types of memory: RAM, EEPROM and ROM. The RAM is only used to store intermediate results during calculation. The EEPROM holds private cardholder values such as a private encryption key or a bank account number. The ROM is used to store the program that runs on the smart card. Smart cards are connected via five pins to the smart card reader. So these systems are "on" only if they are inserted into a reader.

The size of the die is constrained to 25 mm^2. Therefore memory space is hard limited. In average, those systems contain 4 to 20 Kbytes of ROM, 0.1 to 1 Kbytes of RAM and up to 10 Kbytes of EEPROM.

Clock frequency is typically about 3.57 to 5 MHz and an external clock has to be supplied. Smart cards can be seen as special cases of embedded systems.

SUN proposed a specification for the usage of Java on smart cards [11]. Because of the limited memory on smart cards, SUN had to remove some memory-consuming op-codes and features, like

- string manipulation,
- floating point arithmetic and
- threads.

Neither the types char, float, double and long nor operations on those types are supported. Smart cards also do not support arrays with more than one dimension. Object usage is limited as there is no <clinit>-method.

Besides these obvious modifications there are other restrictions resulting from the behavior of a smart card. Java Card systems are not able to load classes dynamically. All classes used are masked into the ROM of the card during manufacturing. Installing through a secure installation process after the card has been delivered to the smart card producer is possible too. Programs executing on the card may only refer to classes which already exist on the card as there is no way to download classes during the normal execution of application code. For more details see [11].

Due to the lack of memory space on smart cards the JVM has to be split into two parts, one for offline preparation as loading, resolving, and linking all classfiles and the other one for online execution of the Java byte code as shown in Figure 1.

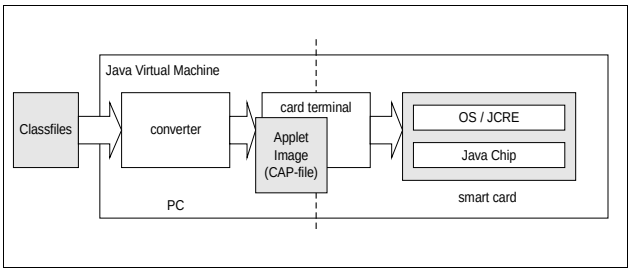


Figure 1. Separation of the Java Virtual Machine into two parts

During preparation the applets are converted to an applet images which can be directly executed on the smart card.

In conventional JVMs the byte code is verified before it is executed. In the smart card area the Byte-Code–Verifier is part of the offline block due to its time and area consumption. Therefore, a downloaded applet is not verified during runtime. To avoid illegal operating applets each applet is signed with a digital signature (CAP-file) so the card itself can determine whether the applet belongs to an environment it trusts or not.

But even a signature is no guarantee that an applet is always working correctly [10]. Unfortunately, there is no way to protect the card against transitive Trojan Horses but currently no such attacks are known. More details on Java Cards can be found in [16], [17] and [18].

3. Simulation-model based on 8051

The first step in the development process was to build a software version of a Java Virtual Machine according to SUN's JavaCard Specification 2.0, which is suitable for implementation on 8051-processor systems [5]. It is a cleanroom implementation and was used to understand the basic behavior of the JVM.

Since the goal was a Java processor some techniques which are used for describing hardware systems were used to implement the Java Virtual Machine. E.g., the execution engine of the JVM has one big "switch-case"-construct. It is figured out that such a switch-case-construct is not very well suited for a small footprint implementation. The next step was to identify groups of opcodes so that function calls like *ALU(opcode, A, B)* or *stackOp(opcode, value)* can be used. Thereby we were able to encapsulate those function-blocks.

The advantage is due to the fact that the simulation model now looks similar to a structural description of a JSM so that the HW-designer just has to make minor modifications only.

On the other hand the disadvantage is that function calls result in a small software overhead.

To avoid nondeterministic and time-consuming searches in classes, methods, or fields in the constant pool it is recommended to have direct access to these structures. This can only be achieved if the addresses of all accessible objects are known in advance. Since resolved applets contain each class they need, each relative location is fixed. But instead of storing an absolute address an applet-relative address is stored. In this way applets can be moved inside the persistent memory (relocatible applets) for the cost of an additional adder.

Firmware or device specific software is located in the application programming interface (API). On smart cards

the API-functionality is located in ROM and the applets are stored in EEPROM. To select an API or applet-method, the highest bit in the available address space is used as a selector (see Figure 2). This bit must be set by the converter while linking the class-files. The base address of the current active applet is stored in an applet base register. Native methods for direct hardware access are also located inside the API. These methods are necessary since different applets must be able to read data, e.g. a bank account number, or they have to increment the number of tries made to activate the smart card.

The implementation requires about 14 Kbytes of ROM on a 8051-system including the cost for the additional modularity of about 3 Kbytes. Since this is a simulation model we have not optimized the source code for the specified architecture.

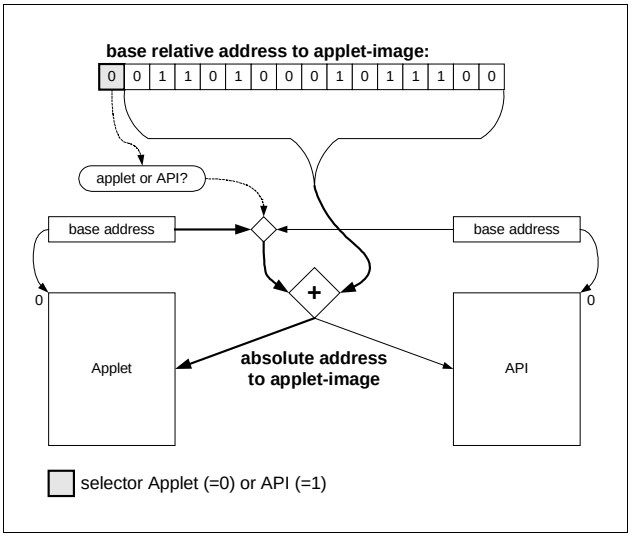


Figure 2. Access to applet-image or API

4. Moving from software to hardware

A Java Card–system is more then just a Java processor. Moreover, the virtual machine is part of the Java Card runtime environment (JCRE). The API, the executive (for handling different applets), and the native methods also belong to the JCRE.

Because of security reasons Java does not offer any byte code for direct hardware access.

To access IO using a Java Virtual Machine on standard or dedicated processor architectures, an API-function is called. Inside this function the Java-environment is left to access IO with the underlying processors IO-opcode, e.g. *mov port_adr, #val* (see Figure 3).

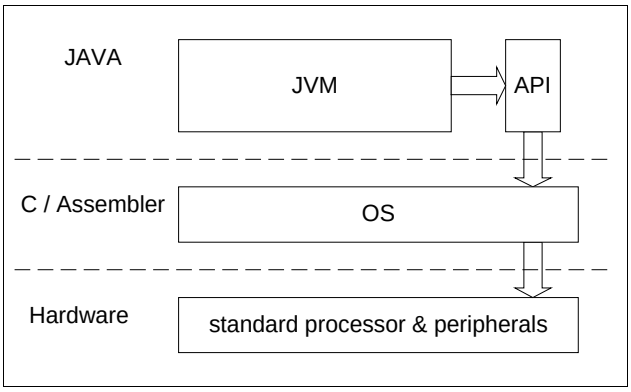


Figure 3. IO-access with JAVA

By using a Java processor there is no environment which could be left. To overcome the IO-access problem in Java the instruction set will be expanded. This is possible since two opcodes ($FE, $FF) in the opcode space are reserved for custom usage.

Due to a missing standard these solutions are proprietary. On PicoJava-II about 33% of the implemented opcodes can not be found in the Java specification.

Obviously, these new (hidden) opcodes can not be generated using an ordinary Java-compiler. Accessing these opcodes becomes somewhat difficult because a processor specific compiler has to be used.

Moreover, applets compiled in such a way are no longer interchangeable and one of the major benefits of Java gets lost.

As soon as implementation details become public knowledge, it should not be too hard to write malicious code, i.e. Trojan horses. This can not be accepted in the smart card area (revealing PIN's and POS's).

It can be shown [9] that only two additional opcodes are necessary: IO-Read and IO-Write.

In the proposed JSM the address of the opcode is traced to avoid illegal IO-access. Since application-applets are stored in reprogrammable memory the JavaCard–runtime-environment (JCRE) is located in ROM or in a *different* EEPROM. Therefore, it is possible to trace the address of the opcodes to-be-executed and a very small online-checker easily can allow or prohibit the execution of the opcode and may generate an exception in case of a fault [2].

Although IO-access is required in different native functions we only implemented IO-Read and IO-Write. Therefore we have no hardwired native functions and the functionality is realized by software inside the JCRE.

5. Java Silicon Machine

In Figure 4 the basic concept for the JSM is shown. Parts of it are already implemented using VHDL. For multi-applet cards one additional opcode has to be implemented: *set_applet*. It is used for selecting one of the applets and loads the *applet base register* with the corresponding start address of the chosen applet.

We have to adopt parts of the JSM to the new JavaCard specification 2.1, since the format of the downloadable applet is *now* specified and it is of great impact for some parts of the JSM.

The JSM is fully controlled by microcode. Therefore many changes may result in some 'software'-updates [1]. To get maximum independence between the submodules the state machines are just loosely coupled. Once these state machines are started, they run automatically until the end of the requested operation and feed back the result. The internal behavior of one state machine is completely hidden to its connected state machines.

To speed up the proposed architecture it is important to know about the dynamic probability of opcodes in a given applet. In [6] some values are given but they are based on benchmarks for complete JVM/JSM's. In the smart card area the situation is different So we are currently analyzing different applets to get the percentage distribution based on the dynamic instruction count. After the linking process the constant pool just contains constants of type integer. Those constants can only be addressed using the *ldc* or *ldc_w* opcode. So the index into the constant pool can easily be replaced with the constant itself. Consequently, we do not need a constant pool which results in some speed-up and less memory usage. Benchmarking different architectures is difficult, since execution speed depends on the compiler and coding style of an applet. We benchmarked the Dallas iButton by measuring the time to call methods and compared the results with first theoretical values of our architecture.

We assume that the iButton is a software implementation since implementation details are not published. The underlying processor itself is driven by an unstabilized ring oscillator operating over a range of 10 to 20 MHz [15]. Therefore the clock frequency of a iButton is not constant. The comparison shows speed up of 100 against the iButton (clocking the JSM @ 3.5MHz).

Techniques like caching promise some speed up but for the cost of additional hardware [6]. Since die size is limited we do not benefit from the usage of these speed-up techniques.

6. Conclusion and future work

We presented some of our experiences in the development of a JSM for small embedded systems like smart cards. We separate the development process into two parts. The experiences resulting from implementing the JVM on a 8051-system directed us to the proposed JSM-architecture. We described some problems and presented solutions for a JSM in the smart card area. It is planned to complete the architecture until the end of the year ´99. For functional verification an APTIX-system explorer M3PC containing four XCV1000-4 will be used.

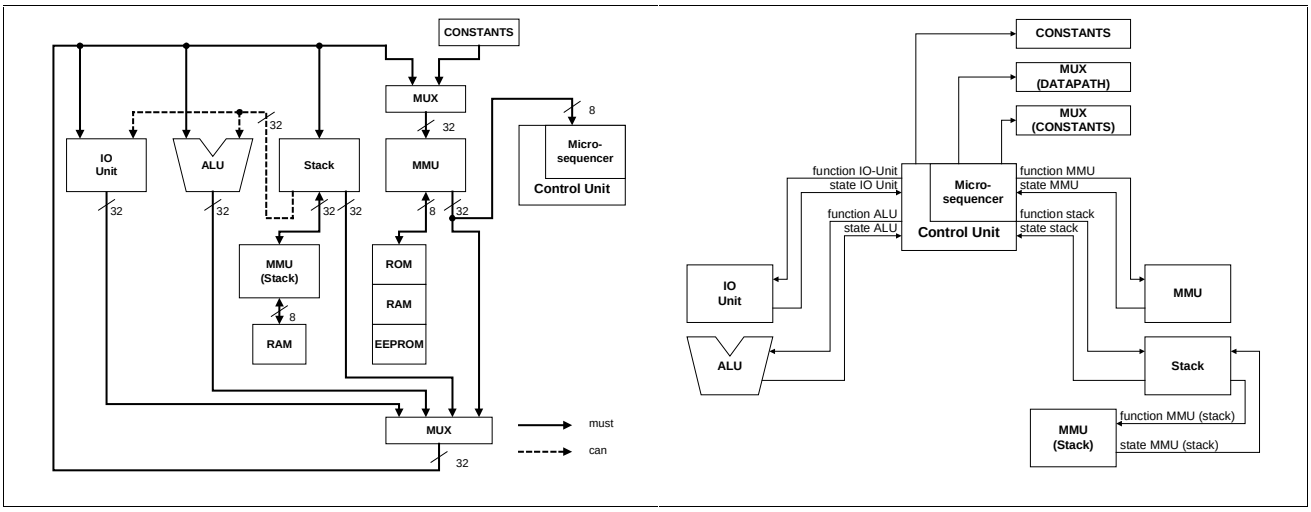


Figure 4. Datapath and control-path of the JSM

7. References

[1] N. Bannow, "*Concept of a Java-Processor,"* Technical Report, University of Rostock, Jan. 1999

[2] F. Golatowski, H. Ploog, R. Kraudelt, O. Hagendorf, and D. Timmermann, "Java Virtual Machines für ressourcenkritische eingebettete Systeme und Smart-Cards," accepted for presentation at JIT'99, Frankfurt, Germany, Sep. 99

[3] C. J. Glossner, and S. Vassiliadis, "The Delft-Java Engine: An Introduction," Euro-Par 97, Conf. Proc., p.766-770, August 1997, Passau, Germany.

[4] http://www.tik.ee.ethz.ch/~jama/

[5] R. Kraudelt, „Entwicklung und Implementierung einer JAVA virtuellen Maschine (JVM) für den Einsatz in besonders ressourcenkritischen Systemen (Smartcards)," diploma thesis, University of Rostock, 1999

[6] Vijaykrishnan Narayanan, "*Issues in the design of a JAVA processor architecture*," PhD-thesis, University of South Florida, 1998

[7] Eric Nguyen, "*JAVA™-Based Devices from Mitsubishi"*, Java One, 1996, slides at: http://java.sun.com/javaone/javaone96/pres/Mitsu.pdf

[8] Patriot Scientific, *Java Processor PSC1000*

[9] H. Ploog, T. Rachui, and D. Timmermann, "*Design Issues in the development of a JAVA-processor for small embedded applications,"* ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA´99, Monterey, Feb 21-23

[10] J. Posegga, and H. Vogt, "*Byte Code Verification for Java Smart cards based on Model Checking,"* 5th European Symposium on Research in Computer Security (ESORICS), Springer Verlag, 1998

[11] JavaCard 2.0 Language Subset and Virtual Machine Specification, http://www.javasoft.com/products/javacard, Sun Microsystems, Inc., 1997

[12] M. O'Connor and M. Tremblay, "picoJava-I: The Java virtual machine in hardware," IEEE Micro, March-April 1997, pp. 45-47

[13] Rockwell, http://www.techweb.com/wire/news/1997 / 09/922java.html

[14] Dallas Semiconductor, http://www.ibutton.com

[15] Stephen M. Curry, "An introduction to the Java Ring," http://www.javaworld.com/javaworld/jw-04-1998/jw-04-javadev.html

[16] Guthery G. B., Java card: Internet computing on a small card, Jan-Feb 1997, IEEE Internet Computing, pp/ 57-59.

[17] Michael Montgomery, "Get a jumpstart on the Java Card", http://www.javaworld.com/javaworld/jw-02-1998/jw-02-javadev.html , 1998

[18] Zhiqun Chen , "Understanding Java Card 2.0," http://www.javaworld.com/javaworld/jw-03-1998/jw-03-javadev.html, 1998

Quantitative Analysis for Java Microprocessor Architectural Requirements: Instruction Set Design[1]

M. Watheq EL-Kharashi Fayez ElGuibaly Kin F. Li

Department of Electrical and Computer Engineering, University of Victoria

P. O. Box 3055, Victoria, BC, Canada, V8W 3P6
{watheq, fayez, kinli} @ engr.UVic.CA

Abstract

Java has emerged to dominate the network-programming world. This imposes certain requirements on its virtual machine instruction set architecture and on any processor design that intends to support Java. The purpose of this study is to carry out a behavioral analysis of the different aspects of Java instruction set architecture. This will help in extracting the hardware requirements for executing Java bytecodes. Recommendations for architectural requirements for Java processors will be made throughout this study.

1. Introduction

Java was introduced to deal with heterogeneous networks, which require building software that is platform-independent [1,2]. This means that a compiled software shipped around the network needs to be able to run on any CPU it lands on (which is formulated as "write once, execute anywhere.") In addition, designed to be a modern high-level language, Java includes all modern features like modularity and object-orientation. To achieve all these goals, Java targets an intermediate virtual platform, instead of direct execution on the host CPU [3,4,5,6]. All we need to execute cross-platform programs on the Internet is to port this virtual layer to the CPU/OS combination we want to run Java on. But, this comes at a high price. The special features supported by Java have a tremendous impact on the overall system performance and impose certain requirements on the Java system [7].

A number of schemas have been proposed to improve Java performance as a tool for programming on the web and networking in general [8,9,10,11,12,13,14]. Some of the promising directions incorporate hardware solutions. Building microprocessors for Java or simply modifying other general-purpose processors to boost Java are among these hardware options [15,16,17,18,19,20,21,22,23,24].

Designing hardware for Java requires an extensive working knowledge about its virtual machine organization and functionality. Java virtual machine (JVM) instruction set architecture (ISA) defines categories of operations that manipulate several data types, reached through a well-defined set of addressing modes [25,26,27,28,29]. JVM specification defines the instruction encoding mechanism required to package this information into the bytecode stream. It also includes details about the different modules required for processing these bytecodes. At runtime, the JVM implementation and the execution environment affect the instruction execution performance. This is manifested directly in the wall-clock time needed to perform a certain task and indirectly in the different overheads associated with executing the job (e.g., memory management) [30].

The goal of this research is to conduct a comprehensive behavioral analysis of the Java virtual machine instruction set architecture [31,32]. Observing the Java instruction set architecture while it is executing Java benchmarks will reveal a lot about the details of the Java environment. This will be reflected in the form of suggestions for the actual hardware improvements and additions to boost the performance of Java. Revised encoding formats and devised hardware organizations (including a certain level of parallelism, pipelining, caching, functionality, ... etc.) with insights about the internal details of Java will lead to better performance. Our rationale for conducting such a study is based on the simple observation that modern programs spend 80-90% of their time accessing only 10-20% of the instruction set architecture [33]. To be most effective, optimization efforts should focus on just the 10-20% part that really matters to the execution speed.

ISA study is of great importance for every attempt to devise a certain arrangement to boost Java performance. The results collected here affect the way of encoding Java instructions into a binary representation for execution by any CPU supporting Java. It also affects the internal processor datapath design for any architecture that targets Java. It is worth noting that although JVM ISA shares many general aspects with traditional microprocessors, it has its distinguishing features. This stems from the fact that it is an intermediate layer for a high-level language.

1. This research was supported through a grant from the National Sciences and Engineering Research Council of Canada (NSERC) to the second author.

For example, the branch prediction model of the underlying hardware affects the overall Java performance, which is the case of all modern microprocessors. On the other hand, JVM-supporting hardware will be unique in the way its stack model handles method invocations.

To undertake these objectives, a Java interpreter was instrumented to produce a Java trace. Pendragon Software's CaffeineMark 3.0 was selected as the benchmark as it is computationally interesting and exercises various JVM aspects [34]. It is a synthetic benchmark that runs 9 tests to measure Java performance. The machine used in the evaluation is an UltraSPARC I/140 running at 143 MHz with 64 Mbytes memory. The OS is Solaris 2.6. Based on the data gathered, general requirements for Java processors are drawn. In doing this study, we followed the methodology used by Patterson and Hennessy in studying the instruction set design [33].

This paper is organized as follows: Section 2 analyzes access patterns for data types. Addressing modes are studied in Section 3. Section 4 is concerned with the different instruction encoding aspects.

2. Access patterns for data types

Here we study the access patterns for different data types. Data types that are heavily used need more attention in case of designing certain hardware architecture to support Java [30]. This information will prove useful when decisions are made about storage allocation.

2.1. Single-type operations

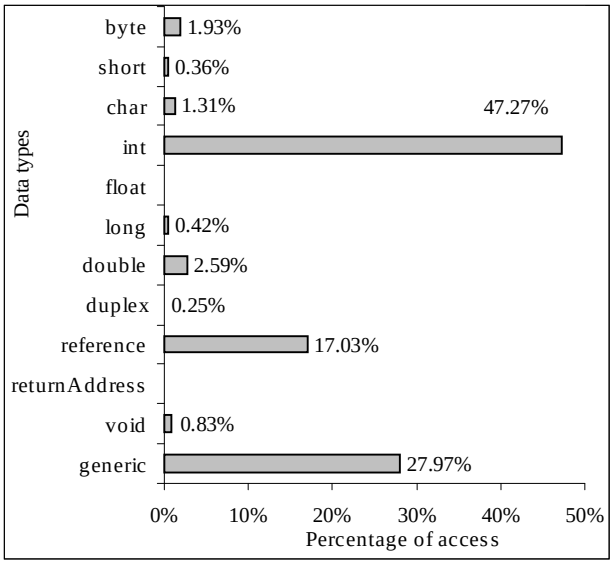


Figure 1. Distribution of data accesses by type.

Figure 1 shows the distribution of data accesses by type. ("Generic" refers to operations that have no data type associated with them.) From this figure we see that integer data types dominate the typed operations, followed by the reference ones. Architectural support for Java object-orientation therefore needs to give privileges to integer and reference data types in hardware. Also from this figure, we see that 32-bit data types are used the most. This will have an impact on the size of the register file and CPU datapaths. Furthermore, a superscalar design may want to provide multiple functional units that process integer and reference data types. From the ALU point of view all integer operations need efficient support.

2.2. Type conversion operations

JVM has a set of instructions that converts data of a certain type to another. This is necessary for a strongly typed language like Java. Figure 2 shows that the conversion from integer dominates all type conversion operations (especially to character.) This information combined with the results from the previous subsection implies that integer conversion operations are the most used. For better Java performance, the ALU design needs to perform this conversion in one clock cycle or less.

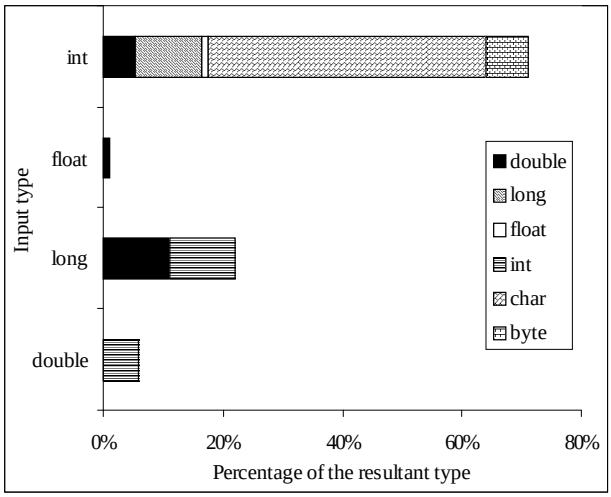


Figure 2. Frequency of type conversion instructions.

3. Addressing modes

This Section is concerned with the use of the JVM addressing modes. The traditional concept of addressing modes, as used in general-purpose processors, is not exactly applicable to JVM, which uses a stack-based intermediate language. This, together with the object-orientation approach, is reflected in the combination of traditional and non-traditional addressing modes [30].

The result of addressing mode usage patterns is shown in Figure 3. Local variable access dominates all other modes. Also, of importance are the immediate access and the quick reference. (The "Quick Reference" item summarizes the quick bytecode optimization.) Hardcoded addressing modes (in which the operand value is encoded in the instruction itself) occupy more than one third of the total addressing modes used. We conclude here, that Java processors need to support at least immediate, local variable, quick referencing, and stack addressing modes.

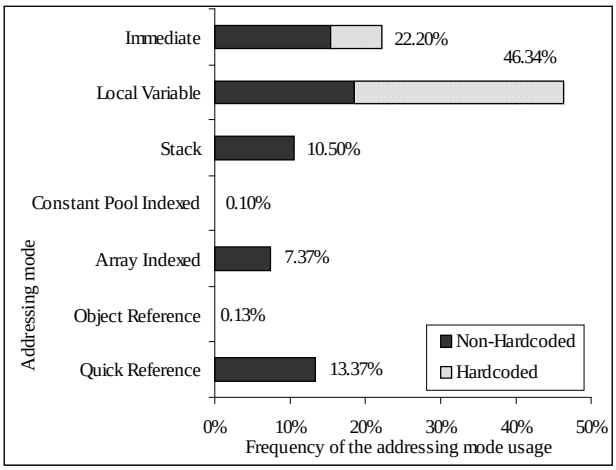


Figure 3: Usage of memory addressing modes.

4. Instruction encoding

JVM specifications require Java bytecodes to be provided as a stream of bytes grouped in variable-length instructions. However, it hardly mentions a general instruction format for the adopted instruction [1]. This irregular format might stand against the generality and efficiency of hardware execution of Java bytecodes.

This section quantitatively analyzes the different fields that constitute JVM instructions. We aim at determining the optimum number of bits required for encoding these fields. The analysis presented here should not be considered contradicting the JVM specification that tells exactly the size of each instruction field. Architectures that provide support for Java might select to have a native instruction format that is different from the JVM one in the addressing modes, data types, etc. This approach will help attaining generality and efficiency.

4.1. Immediates

As a stack machine, JVM does not rely a lot on immediates for ALU operations. Immediates in JVM are either pushed on the stack or used as an offset for a control flow or table switching instruction and could have a length of up to 32 bits. As shown in Figure 4, immediates used in CaffeineMark are up to 18 bits in length with an average of 4 bits (standard deviation of 2.07 bits.) The peak occurs at 3 bits. Three bits are enough to cover more than 50% of immediate usage and 5 bits can cover more than 75%. Based on the statistics shown in Figure 4 we suggest using 8 bits to encode immediates values in JVM instructions, which will cover 98% of the cases. Situations that will require more bits can be handled by the compiler using a special wide format.

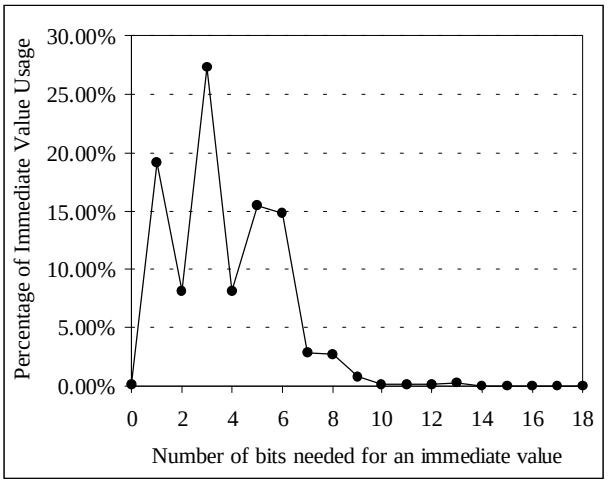


Figure 4: Distribution of number of bits in immediates.

4.2. Array indices

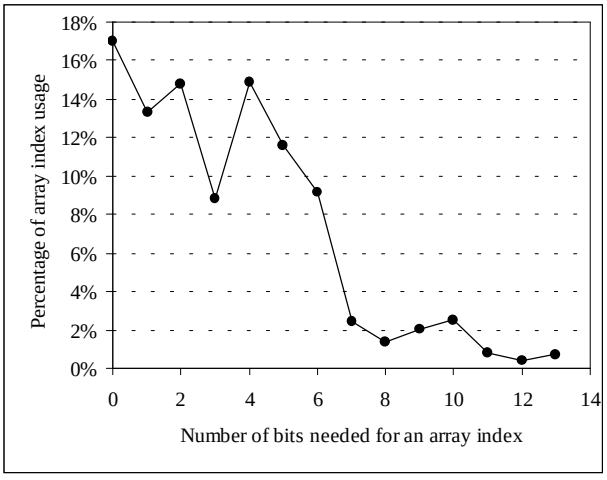


Figure 5: Number of bits representing an array index.

Java technology carries array information down the hierarchy to the JVM. At runtime, the index required to access an array is popped from the operand stack. Although the index can be up to 32 bits, Figure 5 shows that CaffeineMark does not require more than 13 bits to

access arrays (with an average and standard deviation of 3 and 2.79, respectively.) For instruction encoding, we see that 3-bit index size covers more than 50% of the array accesses and 5 bits cover more than 80%. The graph shows an interesting pattern: the maximum occurs at 0 then follows a decaying behavior with a sudden drop after 6 indicating that about 90% of the array accesses take place in the first 64 element. This could give useful guidelines in data caching—the first few elements (up to the 64th) of an array should be cached. This could also have an impact on Java processor's cache organization, such as block size, replacing strategies, etc.

4.3. Constant pool indices

JVM constant pool is a collection of all the symbolic data needed by a class to reference fields, classes, interfaces, and methods. A constant pool index size is either up to 16 bits, or 32 bits if it is in a wide format. From Figure 6, we see that 16 bits cover almost all constant pool accesses (the average is 8 and the standard deviation is 2.9.)

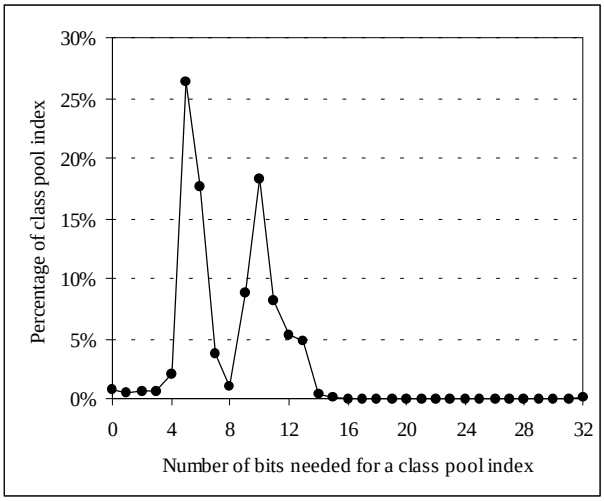


Figure 6: Number of bits for a constant pool index.

4.4. Local variable indices

As mentioned before, instead of specifying a set of general-purpose registers, JVM adopted the concept of referencing local variables. As Figure 7 shows, Java methods typically require up to 16 local variables (with an average of 2 and standard deviation of 1.31), though the specifications allow referencing up to 255, or 65535 in case of wide instructions. The graph also shows nearly no preference in accessing these variables. In designing hardware support for Java, this graph suggests allocating the local variables on-chip. In this case, general-purpose register file can be configured to work as a reservoir for local variables, allowing Java programs to run faster.

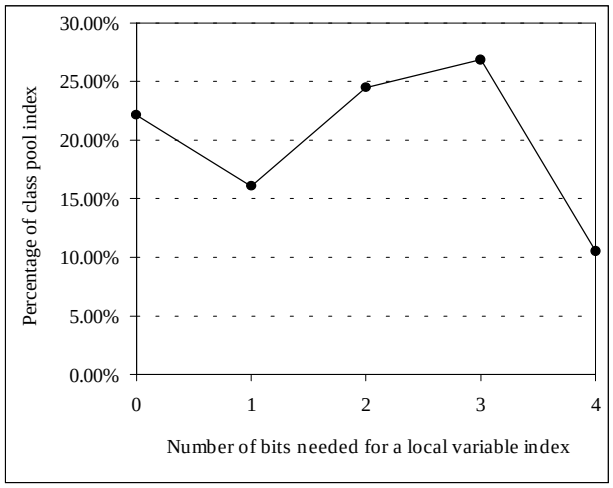


Figure 7: Number of bits for a local variable index.

4.5. Branching distances

Java bytecodes only deliver the offset in branches and JVM converts it internally to the corresponding absolute address. As Figure 8 (dots) shows, CaffeineMark requires a target offset width of less than 10 bits (with an average of 4 and standard deviation of 1.49), though up to 16 bits are allowed. In the design of an instruction format, 8 bits appear enough to cover more than 98% of the offset distances. Furthermore, if a branch target buffer (BTB) is used for branch speculation, a size of 512 bytecodes (256 forward and backward) is sufficient. Figure 8 (triangles) also shows statistics of absolute jump-to address. Although this information depends on the run time environment, it does indicate a typical behavior.

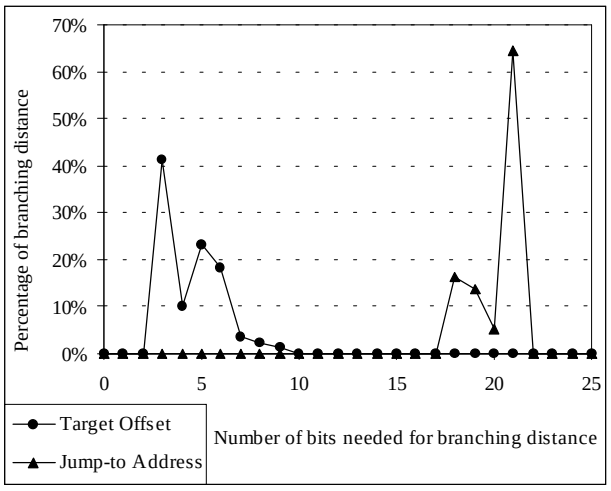


Figure 8: Number of bits for an address.

5. Conclusions

In this work we conducted a behavioral analysis of Java virtual machine instruction set architecture. In the light of the results collected from each part, we drew conclusions about the general architectural requirements for designing microprocessors that support Java. Our study clearly shows that the advanced features of Java are its weakest points in terms of performance. Hardware support is required to increase the efficiency of Java.

Acknowledgement

The authors would like to acknowledge Sun for the Java development kit license. In addition, we would like to thank Kenneth Kent from the Department of Computer Science, University of Victoria, Canada for his support while compiling the Java development kit.

References

[1] J. B. Gosling, B. Joy, and G. Steele, *The Java Language specification*, The Java Series, Addison-Wesley, Reading, MA, 1996.

[2] J. Gosling and H. McGilton, "The Java Language Environment, *A White Paper*," Sun Microsystems, Mountain View, CA, October 1995.

[3] J. Gosling, "Java Intermediate Bytecodes," *ACM SIGPLAN Notices*, January 1995, pp. 11-118.

[4] M. Lentczner, "Java's Virtual World," *Microprocessor Report*, March 25, 1996, 8-11,17.

[5] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, The Java Series, Addison-Wesley, Reading, MA, 1997.

[6] J. Meyer and T. Downing, *Java Virtual Machine*, O'Reilly and Associates, Inc, Sebastopol, CA, 1997.

[7] P. Wayner, "How to Soup Up Java: Part II," *Byte*, May 1998, pp. 76-80.

[8] B. Case, "Implementing the Java Virtual World," *Microprocessor Report*, March 25, 1996, pp. 12-17.

[9] P. Halfhill, "How to Soup Up Java: Part I," *Byte*, May 1998, pp. 60-74.

[10] C.-H A Hsieh, W.-M. W. Hwu, and M. Conte "Compiler and Architecture Support for Java," *a Tutorial presented in the ASPLOS-VII Conference*, Boston, October 1-5, 1996.

[11] C.-H A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhall, and W.-M W. Hwu, "Optimizing NET Compilers for Improved Java Performance," *IEEE Computer*, June 1997, pp. 67-75.

[12] C.-H A. Hsieh, J. C. Gyllenhall, and W.-m. W. Hwu, "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results," *Proc.of the 29th Annual International Symp. on Microarchitectures (MICRO-29)*, IEEE Computer Society Press, Los Alamitos, CS, December 2-4, 1996, pp. 90-97.

[13] H. McGhan and. J. M. O'Connor, "**picoJava**: A Direct Execution Engine for Java Bytecode," *IEEE Computer*, October 1998, pp. 22-30.

[14] J. M. O'Connor and M. Tremblay, "**picoJava**-I: The Java Virtual Machine in Hardware," *IEEE Micro*, March/April 1997, pp. 45-53

[15] B. Case, "Java Virtual Machine Should Stay Virtual," *Microprocessor Report*, April 15, 1996, pp. 14-15.

[16] B. Case, "Java Performance Advancing Rapidly," *Microprocessor Report*, May 27, 1996, pp. 17-19.

[17] M. Watheq El-Kharashi and F. ElGuibaly, "Java Microprocessors: Computer Architecture Implications," *Proceedings of the PACRIM'97*, Victoria, BC, Canada, August 20-22, 1997, pp. 277-280.

[18] R. Grehan, "JavaSoft's Embedded Specification Overdue, but Many Tool Vendors aren't Waiting," *Computer Design* April 1998, pp. 14-18.

[19] Patriot Scientific Corporation, PSC1000 Microprocessor home page, http://www.ptsc.com/psc1000/

[20] M. Tremblay and J. M. O'Connor, "**picoJava**: A Hardware Implementation of the Java Virtual Machine," *Hotchips Presentation*, 1996.

[21] T. Turley, "MicroJava Pushes Bytecode Performance," *Microprocessor Report*, October 28, 1997, pp. 28-31.

[22] T. Turley, "Most Significant Bits," *Microprocessor Report*, August 4, 1997, pp. 4-5, 9.

[23] T. Turley, "Sun Reveals First Java Processor Core," *Microprocessor Report*, November 17, 1996, pp. 9-12.

[24] P. Wayner, "Sun Gambles on Java Chips," *Byte*, November 1996, pp. 79-88.

[25] Sun Microelectronics, "Sun Blazes Another Trail – Introducing the **microJava** 701 microprocessor," *Press Releases*, October 1997.

[26] Sun Microelectronics, "The Burgeoning Market for Java Processors. Inside The Networked Future: The Unprecedented Opportunity for Java Systems," *white paper 96-043*, October 1996.

[27] Sun Microelectronics, "Sun Microelectronics' **picoJava**-I Posts Outstanding Performance," *white paper* 0015-01, October, 1996.

[28] Sun Microelectronics, "**picoJava**-I Microprocessor Core Architecture," *white paper 0014-01*, October, 1996.

[29] A. Tanenbaum and J. Goodman, *Structured Computer Organization*, Fourth Edition, Prentice Hall, Englewood Cliffs, NJ, 1999.

[30] B. Venners, *Inside the Java Virtual Machine*, McGraw-Hill, NY, 1998.

[31] M. Watheq El-Kharashi, F. ElGuibaly, and K. F. Li, "Hardware Adaptations for Java: A Design Space Approach," *Technical Report ECE-99-1*, Department of Electrical and Computer Engineering, University of Victoria, January 25, 1999.

[32] M. Watheq El-Kharashi, F. ElGuibaly, and K. F. Li, "Architectural Requirements for Java Processors: A Quantitative Analysis," *Technical Report ECE-98-5*, Department of Electrical and Computer Engineering, University of Victoria, November 9, 1998.

[33] D. A. Patterson and J. L. Hennessy, *Computer Architecture A Quantitative Approach*, second edition, Morgan Kaufmann Publishers, Inc. San Francisco, CA, USA, 1996.

[34] Pendragon Software, CaffeineMark Benchmark Home Page http://www.pendragon-software.com/pendragon/cm3