JMTP: An Architecture for Exploiting Concurrency in Embedded Java Applications with Real-time Considerations

Rachid Helaihel and Kunle Olukotun
Computer Systems Laboratory
Stanford University, Stanford, CA 94305
{rashhel, kunle}@stanford.edu

Abstract

Using Java in embedded systems is plagued by problems of limited runtime performance and unpredictable runtime behavior. The Java Multi-Threaded Processor (JMTP) provides solutions to these problems. The JMTP architecture is a single chip containing an off-the-shelf general purpose processor core coupled with an array of Java Thread Processors (JTPs). Performance can be improved using this architecture by exploiting coarse-grained parallelism in the application. These performance improvements are achieved with relatively small hardware costs. Runtime predictability is improved by implementing a subset of the Java Virtual Machine (JVM) specification in the JTP and trimming away complexity without excessively restricting the Java code a JTP can handle. Moreover, the JMTP architecture incorporates hardware to adaptively manage shared JMTP resources in order to satisfy JTP thread timing constraints or provide an early warning for a timing violation. This is an important feature for applications with quality-of-service demands. In addition to the hardware architecture, we describe a software framework that analyzes a Java application for expressed and implicit coarse-grained concurrent threads to execute on JTPs. This framework identifies the optimal mapping of an application to a JMTP with an arbitrary number of JTPs. We have tested this framework on a variety of applications including IDEA encryption with different JTP configurations and confirmed that the algorithm was able to obtain desired results in each case.

1 Introduction

With increased demand for more intelligent electronic consumer products, a trend in the embedded system market is emerging: the evolution of embedded systems from fixed functionality environments to systems that can be upgraded with new functionality. One example of such systems is a smart card that changes its encryption algorithm as the need arises. Another example is a personal communication device such as a cellular phone that is able to provide new telecommunication services to the customer. These systems demand functional flexibility that can only be furnished by a programmable architecture. Another feature of this trend is the demand for these systems to communicate with other devices in their vicinity or access remote servers and information databases. This demonstrates the need for a reliable, net-centric infrastructure. This connectivity provides these systems with the ability to upgrade functionality in an automatic and transparent manner. Java has gained great success in one net-centric domain, the web, and has potential to be a front-runner in the new generation of embedded systems because of several key advantages. Java is object-oriented, platform independent, and secure [5]. But more importantly, Java possesses cleaner semantics resulting in more robust code that requires less time to debug. It also makes possible more aggressive static analysis of application code. This directly improves the ability of automatic tools to identify the optimal mapping between a particular application and a specific target architecture. Also, Java is a concurrent language which is important for embedded system specification and design.

Despite its advantages, Java has two problems that restrict the set of embedded applications in which it can be used. The first problem is that of limited runtime performance due to the interpretation overhead from executing the Java Virtual Machine (JVM) on top of a microprocessor [9]. The second problem with Java is the inability to accurately predict or control execution behavior. Embedded systems often have real-time constraints on certain tasks. But Java runtime behavior is very difficult to predict due to the layers of interpretation on top of the real machine, dynamic linking of code, and the interference of such non-deterministic operations as garbage collection.

The solution to these problems must have the following characteristics. First, it has to avoid a drastic increase in system cost. In embedded systems, die area is at a premium; thus, the solution has to trade-off performance increase and runtime predictability with hardware cost. Second, the solution has to be JVM specification compliant to support arbitrary Java code. Third, it must have an easy to use programming environment. Such an environment is based on an automatic compilation process that can efficiently explore the optimal mapping from the application domain to the underlying architecture.

In order to address the performance limitation problem, the solution must exploit concurrency in the target application,

0-7803-5832-X /99/$10.00 ©1999 IEEE.

at least the concurrency expressed by the designer. In Java, concurrency is specified using the *Thread* class. These coarse-grained threads can be exploited using a shared memory multiprocessor architecture and distributing the JVM across the various processors. Although this solution addresses the performance limitation problem, it does not address the runtime predictability problem. However, it is clear that the communication model and cache coherency schemes used in such architectures can only compound the problem. An alternative multiprocessor architecture that addresses this problem is to keep one processor running the JVM while simpler, more predictable processing elements run Java threads.

This paper presents the Java Multi-Threaded Processor (JMTP). The proposed architecture couples a general-purpose processor core with a set of Java Thread Processors (JTPs) and executes a multi-threaded implementation of the JVM. Aside from exploiting thread-based coarse-grained concurrency, the thread processors are designed to enhance performance in other directions. First, the thread processors implement the Java bytecode instruction set architecture and, hence, require no interpretation overhead. Second, the thread processors can exploit certain fine-grained concurrency in threads by adding on specialized function units.

The JMTP architecture is designed to support soft real-time threads executing on the JTPs by enabling the designer to specify timing constraints through Java source code annotations. This is achieved through design for predictability in hardware and aggressive static analysis in software. Primarily, the design guideline is to keep the JTP as simple and as statically predictable as possible by avoiding complex JVM constructs in JTP thread execution. However, dynamic thread control flow affects the predictability of JTP shared resource utilization and scheduling. To avoid sacrificing performance for predictability in such situations, we have added hardware support for adaptively attempting to satisfy timing constraints. This hardware implements an adaptive technique that assigns dynamic priorities to threads based on the dynamic probability of the thread violating a timing constraint. Moreover, in the case where a timing violation occurs, the corresponding JTP would interrupt the JVM for appropriate handling. The complementary software framework would leverage the predictability of the hardware architecture to identify the satisfiability of timing constraints and warn the designer at compile-time if necessary. Notably, the software is able to conservatively guarantee hard real-time constraints in certain cases with deterministic control flow by fixing the resource scheduling.

Our solution does not provide a real-time JVM implementation, but uses the proposed architecture's predictability to provide real-time thread support. The extent of the static analysis does not account for complex JVM operations such as garbage collection or dynamic memory allocation. Thus, the analysis is restricted to code that would execute on a JTP. Hence, a JMTP compiler is unable to evaluate constraints that cannot be completely mapped to a JTP. However, such shortcomings can be addressed by using real-time JVM solutions such as PERC [11].

The rest of the paper is organized as follows. Section 2 introduces the JMTP architecture while Section 3 describes the associated software framework. In Section 4, we describe our experimental methodology and provide results. Finally, we present our conclusions in Section 5.

2 JMTP Architecture

2.1 Overview

The JMTP architecture is a single-chip simple shared-memory multiprocessor as shown in Figure 1. The JTP is a

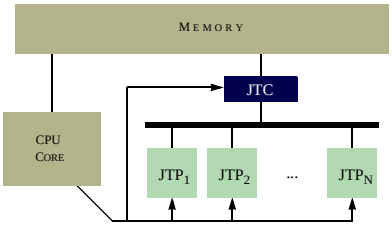

Fig. 1. JMTP block diagram

hardware implementation of the JVM thread specification. Thus, the JVM running on the CPU core views and manages code forked to a JTP as a special case of a regular Java thread. The abstraction used within the JTP and provided to the designer is a subclass of the *Thread* class called *HWThread*. *HWThread* encodes operations like loading the JTP state space by overloading *Thread* native methods. In conjunction, extra functionality needs to be added to the JVM thread implementation to maintain status and manage overall thread control and synchronization. Communication between the JVM and JTPs is handled via memory mapped control/status registers. Arbitration over shared JTP resources such as the shared memory port and hardware synchronization primitives is handled by the Thread Controller (JTC). Priorities assigned by the JTC to individual JTPs can be either fixed for statically deterministic threads or adaptively assigned. Mapping to this architecture is handled by a compilation process that identifies Java tasks to run on a JTP and explores the design space to find an optimal mapping of tasks to JTPs.

2.2 Java Thread Processor Architecture

The Java Thread Processor implements a subset of the JVM specification conceived to strike a balance between hardware complexity and the ease of mapping arbitrary Java code to a JTP. This trade-off is possible because functionality which cannot be mapped to a JTP can execute on to the CPU core-bound JVM. For example, complex JVM memory management and garbage collection is solely handled by the core; thus, all thread memory should be pre-allocated prior to dispatching the thread code to a JTP. An example where complex JVM functionality was ported to the JTP is thread synchroni-

zation primitives, implemented through monitors. This was important to avoid a performance bottleneck if every monitor lock or unlock operation had to go through the core. Another trade-off involved the JTP instruction set. By examining code segments of several target applications we were able to identify a subset of the Java bytecode instructions to support in the JTP instruction set. Additionally, extensions were added to the latter subset to support non-JVM related implementation functionality. These are necessary for setting and monitoring JTP state information.

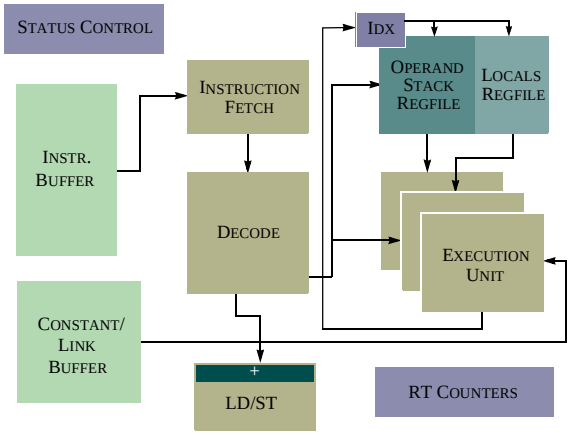

Fig. 2. JTP architecture

The JTP architecture as shown in Figure 2 is fundamentally a 4-stage pipelined RISC-style processor. Although the Java bytecode instruction set is a stack-based architecture so as to reduce code size, a register-register instruction set architecture is more efficient because it does not require the overhead of stack manipulation instructions. However, the JTP emulates a stack instruction set using an annotation to the register file called the stack index register whose two source operands reference the top two entries in the stack and destination references the top of the stack. Arithmetic and logic instructions may target one of three currently supported execution units: an integer ALU, floating-point adder/multiplier, and reconfigurable interconnect of 2x2 8-bit integer ALUs suitable for multimedia type applications. While the integer ALU is basic component of each JTP on the chip, the other two execution unit types are available to some JTPs and not others.

2.3 Real-time Considerations

A key consequence of the reduced complexity of the JTP architecture is runtime predictability. However, access to shared resources such as the memory bus/port or hardware monitor registers is dependent on runtime arbitration decisions. Fixed or static scheduling of these resources does provide a predictable mechanism to assess JTP thread runtime behavior. However, such a solution is not suitable for two reasons. First, it is inefficient as resource usage is generally neither uniform nor equal across the JTP threads. Second, it is oblivious to real-time constraints on JTP threads. Thus, the solution has to be a dynamic arbitration scheme that assigns priorities to colliding requests based on real-time constraint situations. Each JTP includes a real-time counter structure which keeps track of designer-set constraints. These real-time counters are also used to adaptively assign priorities for resource allocation. The underlying reasoning is to allow access to the JTP with the most severe real-time constraint condition. The resulting arbitration scheme is shown in Figure 3.

The real-time arbiter identifies priorities based on the content of the *RT counter* as follows. First, it identifies the leading one in the counter and feeds its position to the *Priority Register* (PR). The PR with the lowest value gets acknowledged first. Each time a JTP is serviced, its associated PR is shifted to the left by one bit. This implies a log2 relation between the value in the RT counter and the resulting priority. The contents of the *RT counter* signify different quantities based on three priority modes in JMTP. The ***Cycle Mode*** loads the *RT counter* with the a set number of cycles to complete the thread task. The *RT counter* is thus decremented each cycle. This mode is suitable for directly mapping time constraints on JTP-bound code. In the ***Slack Mode***, the *RT counter* is loaded with a set number of cycles that the particular task can waste before violating its timing constraint. This quantity is more useful than the previous one because it factors out code related cycles. However, unless the code has well-behaved control flow such as statically bounded loops, it is very difficult to precisely estimate slack cycles. If adaptive priorities are not desired, ***Fixed Mode*** assigns static priorities to the JTPs.

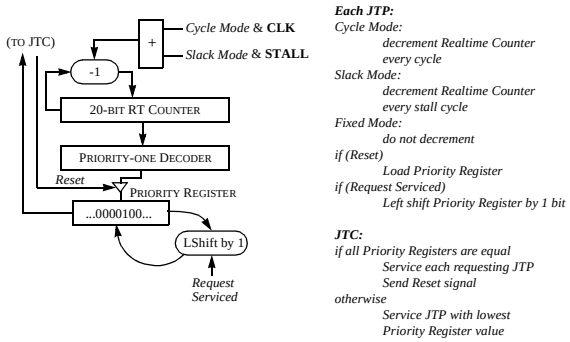

Fig. 3. Real-time arbiter

3 Software Architecture

The JMTP compiler is responsible for analyzing the input Java application to identify the set of concurrent tasks or threads that may potentially be forked as JTP threads and exploring the design space for optimally mapping the application to a JMTP architecture. The latter exploration process is guided by a cost function that weighs the satisfaction of design timing constraints and the overall execution time. A functional outline of the software framework is shown in Figure 5.

3.1 Front-end Analysis

The analysis starts with the static *main* method. For each

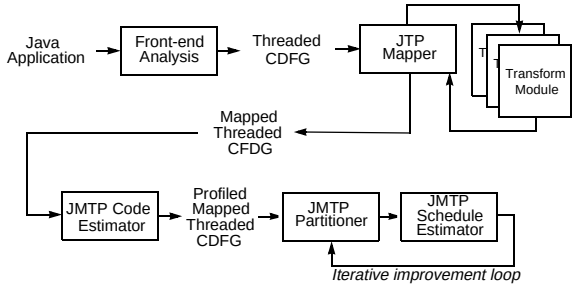

Fig. 5. JMTP software framework

method processed, local analysis is performed to determine local control and dataflow. Next, all methods invoked by the current method are recursively analyzed. Finally, reference points-to values are resolved in order to determine global data dependence information. A more detailed discussion of our front-end analysis technique is found in [7].

The CDFG representation shown in Figure 4 involves two main structures. The first structure is a table of static and pre-allocated class instances. Aside from object accounting information, this table maintains a list of entries per object; each entry represents either a method or a non-primitive type data field. The data field entry is necessary for global analysis because data fields have a global scope during the life of their instances. Arrays are treated exactly as class instances. In fact, arrays are modeled as classes with no methods. The method entries point to portions of the second main structure in the representation. The second structure is the control dataflow information. Its nodes are bytecode basic blocks. The edges represent local control flow between basic blocks within a method as well as global control flow across method invocations and returns.

This CDFG representation also maintains special constructs for identifying and modeling Java threads and bytecode loops. The front-end analysis identifies designer-specified threads by capturing the JVM specification for the Thread class. Thread CDFG information is directly obtained by tracing the flow of the particular thread's *run* method. The representation reserves special edges to denote Java inter-thread communication primitives. Loops are identified through traditional iterative dataflow analysis [1].

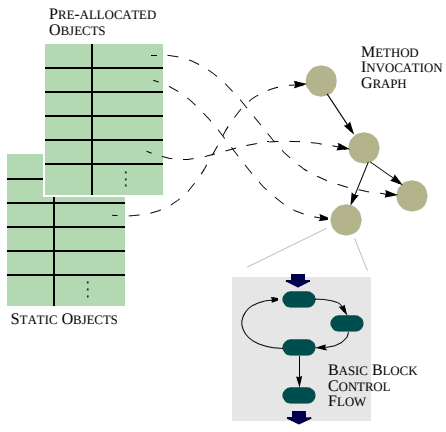

Fig. 4. Front-end analysis CDFG representation

3.2 JTP Mapping

The JTP mapper is responsible for identifying segments of application code that capture the coarse-grained concurrency in the application. The mapper then transforms these code segments to comply with the JTP specification, if possible.

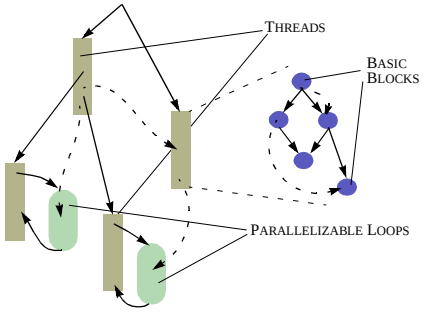

Fig. 6. Concurrent CDFG blocks graph

First, the JTP mapper captures application concurrency into an annotated graph, as shown in Figure 6, built on top of the CDFG. This graph models the control flow of designer-specified Java threads directly from the CDFG. It also captures parallelizable loops by examining true data dependencies in the CDFG. We designate these concurrent structures as *concurrent blocks*. Second, the bytecode for each target thread or loop body is mapped to JTP bytecodes. This process is linear if the original bytecode segment does not violate the subset of the JVM that the JTP implements. If such a violation is detected, the mapper - based on the pattern of the violation - attempts a suite of code transformations to remove the violation. Such transformations vary from simple bytecode replacement to code motion. If no transformation succeeds in eliminating the violation, the target block is restricted to run on the JVM.

3.3 JTP Code Estimation

The purpose of JTP code estimation is to provide execution time guidance for the partitioning phase of the software compilation framework. This is accomplished by a mix of static code analysis aided by dynamic profiling data. Dynamic profiling information is achieved by instrumenting the target Java runtime environment executing on the target RISC core. Specifically, two major quantities are measured. First, execution timestamps are collected at the boundaries of the concurrent blocks identified by the mapper. Second, control flow profiles of branches in the code are compiled to provide bias to different execution flows in the code.

The static estimation component consists of a partition-independent step performed once prior to partitioning phase and a partition composition step that has to be computed for each

partition instance examined. The reason is due to arbitration on shared resources such as access to memory and monitors. This is clearly dependent on the number of JTPs in operation and the code each is executing.

Pre-partition estimation analysis involves direct JTP code analysis to compute dynamic instruction count for the thread code. The analysis also factors in the effects of JTP pipeline hazards. An average execution time is computed based on dynamic control flow data. Arbiter references are identified and modeled. The tool currently models these reference patterns using a uniform random process. As a post-partition step, the models from all JTPs in operation are composed to model the overall arbiter reference load and, consequently, identify the reference service latency. With that, the JTP code execution time is obtained. It is important to note that this estimation process has to trade-off accuracy and computation speed. The key is that the pre-partition analysis can be as accurate as possible because it is performed once per compilation. Moreover, the model composition step has to execute as fast as possible because it is part of the partitioning loop. This directly affects the choice of reference pattern model. We have examined a more complex time-variant model, but the achieved accuracy did not justify the added computational complexity.

An issue facing the estimation process is determining the execution behavior of dynamically bounded loops and recursive method calls. The baseline approach for dealing with this problem is to rely on dynamic profiling data to approximately bound the execution. Of course, such results are flagged to the partitioner as to provide no guarantee for real-time analysis. An alternative approach which is also used by the estimator is to identify bytecodes that set execution bounds and trace the dataflow of the associated variables across the CDFG. Such variables may be identified as application/dataset parameters. These parameters can be dealt with by either prompting the designer for guidance or, more aggressively, by factoring them into the partitioning process. However, in the latter case, the compiler has to be aware of the added complexity of the partitioning step. Currently, we implement complexity bounds on the partitioner to restrict the number of parameters handled.

3.4 JMTP Partitioner

The partitioning algorithm assigns target concurrent blocks identified by the mapper to execute either on the JVM core or on a JTP so that soft real-time constraints are best met and the overall execution time is minimized. This is achieved in two phases: (1) partition generation and (2) iterative partition improvement. Before describing these phases in detail, the cost function is defined.

The cost metric Q used is an arithmetic mean of timing constraint satisfiability and overall execution time as shown in Figure 7. The quantity τ effectively measures the average difference between a constrained block execution time and constraint values. The exponential relationship between τ and the timing difference translates into a blowup in Q as the possibility of a timing violation increases. So, minimizing Q improves timing constraint satisfiability. Note that estimated timing values are increased by a factor of α to allow for estimation errors. The quantity υ is the ratio of the overall execution times before and after partitioning. The overall execution time is computed by a quick JMTP schedule estimator using the concurrent blocks graph in conjunction with JVM profiling data and JTP code estimator results.

$$Q = K_\tau \cdot \tau + K_\upsilon \cdot \upsilon$$

where $K_\tau + K_\upsilon = 1$

such that

$$\tau = \frac{1}{N} \cdot \sum_{i \in S_c} e^{-K \cdot \frac{t_{c_i} - (1+\alpha) \cdot t_{est_i}}{t_{c_i}}}, \text{ and}$$

$$\upsilon = \frac{ET_p}{ET_0}.$$

S_c = Set of N time constrained blocks.

t_{est_i} = Estimated block execution time.

t_{c_i} = Block time constraint value.

ET_p = Partitioned overall execution time.

ET_0 = Unpartitioned overall execution time.

Fig. 7. Cost function parameters

The first phase of the partitioning algorithm generates an initial mapping of blocks to the JTPs and JVM core. Initially, all blocks are assigned to execute on the JVM core. For each block that the mapper has cleared to execute on a JTP, the algorithm identifies the set of suitable JTPs and assigns the block to the JTP whose schedule introduces minimal execution overlap. The resulting value of Q is computed, and the assignment that provides the best Q value is adopted. The algorithm repeats until no move improves Q.

```
repeat {
    best.move_block = 0
    foreach B not assigned to a JTP {
        if (B contains a JTP violation)
            goto next iteration
        move_to_JTP_id = identify JTP to move B to
        B.Q = compute_Q(overall execution time and
                        time constrained blocks)
        if (B.Q < best.Q) {
            best.move_block = B
            best.Q = B.Q
        }
    }
    assign best.move_block to move_to_JTP_id
} until (best.move_block == 0)
```

Fig. 8. Partitioning algorithm pseudo-code

The second phase of the partitioning algorithm attempts to iteratively improve on the partition generated in the first phase. Each block assigned to a JTP is moved to the JVM core. The partitioner resumes phase one analysis starting with the block-

to-JTP assignments minus the moved block. The partitioning that leads to an improved *Q* value is kept. The process is repeated until no single move can be found to improve *Q*.

4 Experimental Methodology and Results

To evaluate the proposed JMTP architecture, we developed a system simulator, *JMTPsim*, based on the LESS simulation environment. LESS is a simple cycle-accurate single chip multiprocessor simulator [6]. From LESS we leveraged cycle accurate CPU and memory system models. The JTP instances along with the JTC models were coded in C++ and integrated with the LESS environment. We selected *kaffe*, a freely available implementation of the JVM with JIT compilation support [12]. The kaffe code was annotated with modules necessary to support runtime interaction with the JTPs. JMTPsim was used to capture *HWThread* execution profiles. However, JVM execution data was obtained by instrumenting kaffe and running the target application on a 100 MHz *SuperSPARC* processor because kaffe provides an interpreter without just-in-time compilation [4] for the MIPS platform.

In addition to the hardware simulation environment, we developed the *JMTPc* compiler by coding and integrating the steps outlined in Figure 5 into the previously developed Java front-end analysis framework. Experimental results using *JMTPc* to target the *JMTPsim* environment are provided to show the impact JMTP has on improving overall performance and its real-time support capabilities.

4.1 Performance Results

The MPEG decode application was used to demonstrate JMTP's ability to exploit coarse-grained application parallelism exhibited by video stream compression/decompression algorithms [8]. The original Java code obtained from [2] was modified to utilize *HWThread* instances to map to JTPs. Four different JMTP configurations were attempted: (1) a 2-JTP configuration, (2) a 2-JTP configuration with one reconfigurable ALU, (3) a 3-JTP configuration, and (4) a 3-JTP configuration with one reconfigurable ALU. In the 2-JTP configuration cases, one JTP was assigned to do an inverse discrete cosine transform (IDCT) while the other was variable length decoding (VLD) input blocks. In the 3-JTP cases, the third JTP performed a portion of the computation for motion information. The extra reconfigurable ALU is used by the JTP handling the IDCT to enhance the computation. All configurations were tested on a small (450KB) sample MPEG stream which was pre-loaded into JMTP memory prior to collecting performance data. For this experiment, *JMTPc* was not allowed to search for implicit concurrency in the code; this forced the partitioner to map each thread to a JTP as desired.

The results shown in Table 1 indicate an approximate 0.25X speedup per mm^2 of additional JTP processing element. This number is expected to decrease with more JTP elements due to contention on the memory bus. This contention can be easily identified from the cycles-per-bytecode results which increase by half a cycle going from two to three JTPs. However, The 2-JTP case has a smaller speedup per area profile than the 3-JTP case because the 3-JTP case better amortizes the fixed cost of the JTC and bus area.

Configuration	Speedup (SU)	Average cycles per bytecode	HW Cost (mm^2)	Δ(SU) per mm^2
2-JTP	2.2	2.7	5.2	0.23
2-JTP with RU	2.2	2.7	5.4	0.22
3-JTP	2.9	3.2	7.1	0.27
3-JTP with RU	3.0	3.2	7.3	0.27

Table 1. MPEG decoder results (Area estimates are based on a 0.25µ process)

In order to assess the effects of limited memory bandwidth on JMTP architecture scalability, we increased the number of read ports on the memory module to two and three ports. Table 2 reports the resulting occupancy of a memory read for the MPEG decoder application with 3-JTP configuration. As expected the occupancy of load bytecodes decreases with increased memory bandwidth.

Number of Read Ports	Load Occupancy (cycles)	Av. cycles per bytecode
1	6.1	3.2
2	4.6	2.9
3	3.9	2.8

Table 2. Effects of memory bandwidth

Several applications were used to test the software framework and the partitioner in particular. First, we used *raytracer*, a simple graphical application which renders two spheres on top of a plane with shadows and reflections due to a single, specular light source. Another test application we used was a variant of *BYTEMark's* [3] *IDEATest* application. *IDEATest* performs IDEA encryption on a 4000-byte plaintext block. Third was a simple Huffman compression application. Table 3 outlines the analysis results from *JMTPc* prior to partitioning.

Application	*HWThreads*	Parallelizable loops
Raytracer	2	8
IDEATest	3	5
Huffman	2	6

Table 3. Application characteristics

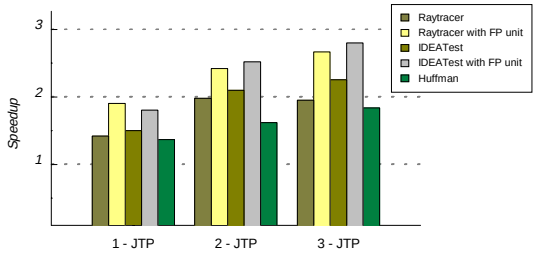

Fig. 9. Partitioning results

Figure 9 shows the results of the design exploration performed by the compiler for different cost alternatives. Further exhaustive analysis showed that the partitioner was able to locate the optimal or near optimal results in each case.

4.2 Real-time Analysis

To demonstrate JMTP's real-time thread behavior, we devised an application, *RTTest*, that uses two JTPs each of which performing IDCT on a single 8x8 matrix. JTC arbitration was set to *fixed mode* with equal priorities to the two JTPs. The thread running on JTP_1 was assigned a timing constraint T_c. The value of T_c was gradually decremented until the JTP_1 signalled a time constraint violation for $T_c = T_{cv}$. Then, arbitration was switched to *cycle mode* whereupon JTP_1 completed without a violation for $T_c = T_{cv}$. Figure 10 plots the change in the number of JTP_1 accesses on the bus sampled across four equal periods of time. While the *fixed mode* case exhibits a constant access pattern, the *cycle mode* increasingly biases the priorities in favor of JTP_1 until its task completes. Note that in the final period, *cycle mode* accesses drop below fixed mode accesses because JTP_1 completes during that period.

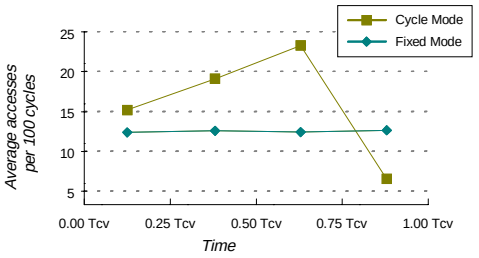

Fig. 10. *RTTest* priority variation

5 Conclusions

We have provided an overview of the Java Multi-Threaded Processor which we position as a solution for a wide range of embedded applications that are targeted to the Java platform but require higher performance or runtime predictability and real-time support. The JMTP architecture incorporates a full JVM running on a general purpose processor with little modification to support the extended hardware architecture. From the designer's perspective, JMTP programming is transparent. JMTP architecture enhances application performance by dispatching thread execution to JTPs. We have shown that JTPs provide appreciable performance improvement given their hardware cost for an MPEG decoder application.

The JMTP architecture includes a compilation framework that optimizes the use of the hardware by identifying implicit as well as expressed application concurrency and exploiting it. This mainly involves a thread partitioner that generates a mapping of the application to the JMTP structures so as to minimize a cost metric combining timing constraint satisfiability and improvement in overall execution time.

Acknowledgments

This work was sponsored by DARPA under grant number DABT63-96-C-0037 and GTRC contract number 98-IT-674. The authors wish to thank the reviewers for their insightful comments. We also wish to thank Lance Hammond for providing and helping with the LESS simulator.

References

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

[2] J. Anders. Java implementation of an MPEG decoder, *http://rnvs.informatik.tu-chemnitz.de/~ja/MPEG/MPEG_Play.html*.

[3] *BYTE Benchmarks* at http://www.byte.com/bmark/bmark.htm

[4] T. Cramer, et al. "Compiling Java Just in Time," in *IEEE Micro*, pp. 36-43, Vol. 17, No. 2, May-June 1997.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[6] L Hammond, et al. "Data Speculation Support for a Chip Multiprocessor," ASPLOS VIII, October 1998.

[7] R. Helaihel and K. Olukotun. "Java as a Specification Language for Hardware-Software Systems," in the Proceedings of the *International Conference on Computer-Aided Design* (ICCAD), November 1997, San Jose.

[8] E. Iwata and K. Olukotun. "Exploiting Coarse-grained Parallelism in the MPEG-2 Algorithm," in the *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture* (HPCA-4), February 1998, Las Vegas.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[10] H. McGhan and M. O'Connor. "PicoJava: A Direct Execution Engine for Java Bytecode," *Computer*, pp. 22-30, Vol. 31, No. 10, October 1998.

[11] K. Nilsen. "Adding Real-time Capabilities to Java," in *Communications of the ACM*, June 1998, Volume 41, Number 6, pp. 49 - 56.

[12] T. Wilkinson. *Kaffe - A Virtual Machine to Run Java Code*, at http://www.kaffe.org.