Scientific Programming 13 (2005) 1–15 1
IOS Press

JIST: Just-In-Time scheduling translation for parallel processors[1]

Giovanni Agosta*, Stefano Crespi Reghizzi, Gerlando Falauto and Martino Sykora
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32 – 20133 Milano, Italy
E-mail: {agosta, crespi, sykora}@elet.polimi.it

Abstract. The application fields of bytecode virtual machines and VLIW processors overlap in the area of embedded and mobile systems, where the two technologies offer different benefits, namely high code portability, low power consumption and reduced hardware cost. Dynamic compilation makes it possible to bridge the gap between the two technologies, but special attention must be paid to software instruction scheduling, a must for the VLIW architectures.
We have implemented JIST, a Virtual Machine and JIT compiler for Java Bytecode targeted to a VLIW processor. We show the impact of various optimizations on the performance of code compiled with JIST through the experimental study on a set of benchmark programs. We report significant speedups, and increments in the number of instructions issued per cycle up to 50% with respect to the non-scheduling version of the JIT compiler. Further optimizations are discussed.

1. Introduction

Parallel computation is a response to the growing need for processing speed, which exists in both embedded and high-performance systems. In the embedded systems field, demanding requirements on power consumption and processing speed are imposed on the design. In the high performance field, processing speed is the main goal.

Parallelism can be employed at different levels of granularity. At fine grain, parallelism is found between individual instructions that do not have data dependency constraints, while at increasingly coarser grain one can find parallelism between different iterations of the same loop, or between different tasks of a system. An attractive technological solution for achieving speed with low power is offered by new VLIW processors designed for embedded systems, but further conditions are imposed by software considerations.

VLIW processors are the result of a trend to move computation from runtime (and the hardware) to the compiler: rather than performing instruction scheduling by means of a specialized hardware support, the compiler is required to provide a scheduled code to the processor.

Currently, most VLIW-based systems adopt a static compiler, but additional requirements imposed by software considerations in our target domain lead us towards a more dynamic setting. Many applications, especially in multi-media and communication, operate with downloadable programs and dynamically linked libraries. The downloaded program can be a industry-standard machine language or a machine-independent bytecode. In both cases the code must be translated to the native VLIW code and scheduled for instruction-level parallelization, in order to obtain the required performances. But code transformations should not be done statically (i.e. ahead of execution) for a number of well-known reasons, including unacceptable compilation delay and unavailability of dynamically linked routines. One is left with the possibility of dynamically translating from the downloaded code to the native scheduled code. But of course many critical factors affect the final convenience of such solution. The hardware must be suitable to run the compiler, the compilation algorithms must combine speed and precision,

[1] This work was partially supported by STMicroelectronics.
*Corresponding author.

ISSN 1058-9244/04/$17.00 © 2005 – IOS Press and the authors. All rights reserved

the optimizations performed must be chosen for their cost/effectiveness. This is the technical challenge we have taken in the long run, to design a dynamic compiler from Java bytecode to the (HP and STM) Lx VLIW processor, that would achieve good performance for typical embedded programs.

The choice of Java as the target language is appropriate for our task, thanks to two qualifying features:

- Java is compiled to a machine independent bytecode, which makes it suitable for downloading new code from a remote source, which has no knowledge of the target architecture;
- Java is a popular language, with a strong support from the industry; it is being favorably considered for adoption is the embedded and mobile systems [1,2] as well as in scientific and numerical computing [3–5].

In both the embedded and the scientific computing areas, the strongest traits of Java are the popularity with software developers, the large number of libraries already available, and most of all the portability features. On the other hand, the main drawbacks are considered to be the reduced performances due to bytecode interpretation and lack of optimization in JIT compilers.

Therefore, more parallelism needs to be extracted by the JIT compilers to make them suitable for adoption in these fields. While existing techniques from the static compilers can sometimes be ported over to the JIT, the risk of unfavorable trade-off between compiler overhead and performance increase is high, and must be analyzed through experimentation. However, there are no implementations of Java bytecode JIT compilers that can be used for evaluation on VLIW machines.

The JIST (Just-in-time Scheduling Translator) project was started to address this issue. As a result, a complete running prototype has been implemented, and a summary of design decisions and measurements can now be reported. They should interest those interested in the performances of Java on a small VLIW processor, and be also valuable to people working on dynamical translators in different settings. We focus on the design aspects that are critical for the intended target architecture: register allocation and instruction scheduling; but we also consider the impact and the management of general issues such as memory disambiguation. The measurements are fairly analytical and allow to pinpoint the cost/effectiveness of individual optimizations.

The analysis is focused on Java code rather than on native libraries. This choice is justified by the fact that libraries are part of the resident system and can therefore be statically compiled and optimized. Moreover, as the application field of Java becomes wider, pure-Java libraries tend to become more desirable than native C libraries, since they are easier to produce for programmers whose primary language is Java itself. The ability of the Java platform to effectively use its native language for expansion of the class library is also important to avoid cluttering the native libraries (which are usually shipped with the Virtual Machine itself) with features relevant only to a limited application domain, while allowing the application designer to use third party libraries within the main application when needed.

The paper is organized as follows. Section 2 presents a brief account of related work. Section 3 describes the architecture of the JIST VM and compiler. Section 4 describes the register allocation and scheduling policies, and the memory disambiguation technique. The experimental results are reported in Section 5, while Section 6 draws the conclusions and mentions future developments.

2. Related works

Early proposals for exploiting instruction-level parallelism (ILP) for Java Bytecode where based on the Java Processor concept [6,7]. Sun Microsystems MAJC processor [7] was designed to efficiently execute Java Bytecodes. The pioneering work of Ebcioğlu et al. [6] proposed the use of a JIT translator coupled with a software scheduling algorithm on a processor that offered instruction level parallelism. The idea was more of designing a new processor for Java Bytecode execution than to use a standard VLIW machine, but it included the possibility of adopting the same algorithm for different architectures. Our basic scheduling algorithm is rather similar, since both are greedy and schedule each instruction as soon as its operands are ready. However, as the goal of [6] is more to propose a novel architecture, their published data ignore the scheduling and code generation overheads for any real architecture. As far as pure scheduling is concerned, JIST matches the ideal performances of [6], and provides experimental results for a real VLIW machine.

Other relevant works deal with general binary compatibility problems. One especially significant work is DELI [8]. The scope of DELI is beyond that of a Java VM/JIT, providing the ability to emulate different hardware machines.

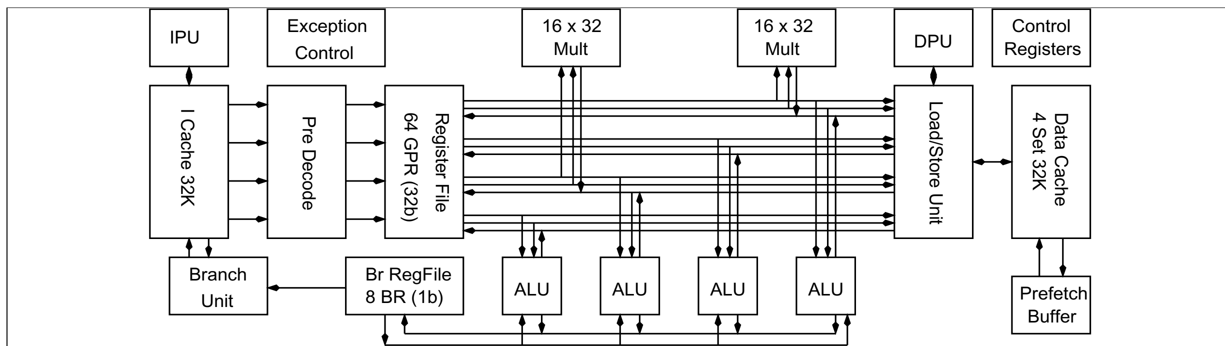

Fig. 1. Lx single cluster architecture.

Of the various Java VMs and JIT compilers [9,10], LaTTe [11] and microJIT [12] are somehow related to our own. They are both based on the Kaffe framework, but they schedule instructions for general purpose RISC rather than VLIW machines. LaTTe is notable for linear scan register allocation, while microJIT does a faster, single pass register allocation.

Focusing on dynamic compilation for VLIW, the DAISY project [13] involved fast binary to binary translation, including optimization passes, like copy propagation and instruction scheduling. However, the source language was the machine code of IBM's PowerPC, and the architecture featured simplifying assumptions such as exception-free execution. DAISY's instruction scheduling is more aggressive than our current implementations, as the scheduled region is a tree region rather than a basic block.

Earlier works in dynamic compilation for VLIW machines focused on binary compatibility between different VLIW architecture as [14], or, as DAISY, on compatibility between traditional RISC machines and VLIWs. On the other hand, our work is oriented to an higher level of code compatibility, as is allowed by machine-independent bytecodes.

Yet other projects, such as [15], introduce architectural variants to the basic VLIW designs.

A survey of dynamic compilation issues can be found in [16].

For the memory disambiguation issue, the work nearest to our field is [17], which provides an extensive list of works related to alias analysis.

3. JIST architecture

The target for the JIST project is the Lx processor family developed by HP and STMicroelectronics [18], a clustered VLIW architecture. Figure 1, adapted from [18], shows a single cluster. The single cluster Lx employed in our project, in addition to the ability to issue up to four instructions per cycle, allows simple predicative execution through *select* instructions.

JIST is based on the Kaffe VM and JIT compiler [19], a clean room implementation of the Java Virtual Machine (*JVM*) specification [20], plus the associated Java class libraries needed to provide a Java runtime environment. It is written mostly in ANSI C, with machine dependent parts in Lx assembly. It offers three main execution modes, or engines, from the less to the most efficient:

intrp A port of the Kaffe interpreter, implements the Java VM specifications;
jit A port of the Kaffe JIT compiler, version 3, produces sequential machine code;
jist The scheduling JIT compiler, adds several scheduling and register allocation options to the standard `jit`.

Figure 2 shows the architecture of the JIST VM: a Java program, compiled to bytecode by an external compiler, is read by the VM frontend and translated in the intermediate representation, *Kaffe IR*. When the VM is in interpreter mode, each Kaffe IR instruction is simulated by the `intrp` engine. The only interaction with the native instruction set happens at native method calls, where the parameters of the call must be translated from the internal representation of Kaffe to the machine call arguments format.

On the other hand, the JIT compilers translate code to the Lx native instruction set. The scheduler pass is only performed in `jist`.

The generated code and the VM itself are executed on an Lx ISA simulator.

The `jist` works in three steps:

1. perform an analysis of the control flow, in order to generate information for the code generation and scheduling pass;

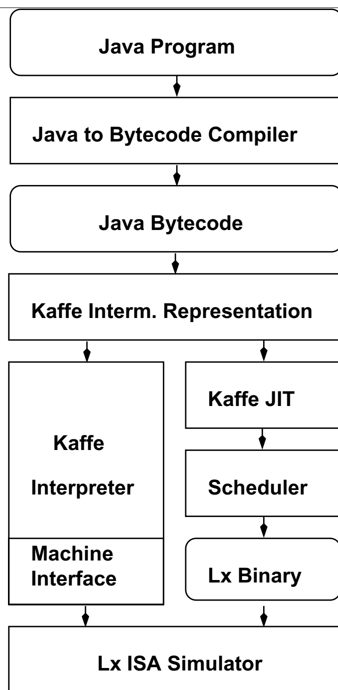

Fig. 2. Execution of Java in JIST.

2. translate each bytecode to native code, performing register allocation at the same time;
3. schedule the translated code, and emit it.

The translation is performed at method grain, that is each method is entirely translated in a single activation of the translator, and the translation takes place when the method is invoked for the first time.

Figure 3 shows how a sequence of Java Bytecodes (the same as in [6]) is translated, first to the Kaffe IR and then to the Lx native code. Finally, the schedule obtained is highlighted by the machine issue cycle.

Figure 3 shows how the Kaffe IR generate separate IR instructions for computation and stack management (see the explicit *pop* and *Stack* access operations). Due to the *register allocation* phase, most copies between Java stack and locals can be removed, so that only relevant computation is actually performed in the generated code. The *scheduler* then allows the issue of multiple instructions in a single cycle, e.g. the two additions in Fig. 3.

The scheduling and register allocation algorithms are the main improvement over the original Kaffe VM, along with the memory disambiguation phase, and will be described in detail in the next section.

4. Dynamic optimization

This section describes the optimization passes we have developed for the JIST compiler, namely register allocation, instruction scheduling, and memory disambiguation.

4.1. Register allocation

Recall that the Kaffe VM offers a two-step register allocation policy. First, it tries to allocate as many variables as possible to global registers. These bindings are guaranteed to hold across basic blocks, so that there is no need for spills at basic block boundaries. Then, short lived variables are allocated to non-global registers with a *least recently used* policy.

The number of global registers is a parameter of the allocation algorithm, and it affects the performances of the translated code as the number of spills needed at the basic block boundary varies. Experiments conducted during the development of JIST proved that 6 to 8 global registers are needed to minimize the spills.

In our case, the Lx architecture provides a large number of registers – 64, minus those used for special purposes, such as the stack pointer. Therefore, even after allocating enough global registers, a large number of registers are available for additional optimization.

Since the Kaffe register allocator does not change an allocation unless it is forced to do so, a considerable number of false dependencies are created by the allocation and code generation phase. In order to avoid a costly post-scheduling reallocation phase, we replaced the Kaffe allocator by a different allocation policy, called *cyclic register allocation*, that allocates a new register every time a value is written. This removes the false dependencies, and gives the scheduler more opportunities for out-of-order execution.

Figure 4 shows how the cyclic register allocation affects the translation flow and the quality of produced code. The first two columns on the left report the Java code of two consecutive instructions and its translation into a simplified version of *bytecode*. For each bytecode instruction it is shown its semantics, as list of operations on the *Java Frame*: for example, the bytecode `add` means that the addition must be performed using the two topmost operands on the Java Frame, removing them from the stack and then pushing the result. On the fourth column, the emitted native code is listed. It can be noticed that each register is associated to a Java Frame *slot*. The arrows explain how the Kaffe classic register allocation policy works: it tries to maintain as

ByteCode	Kaffe IR Macros	Lx Code	Cycle
iload_1	move_int (local > Stack(0))		
iload_2	move_int (local > Stack(0))		
iadd	add_int(Stack(1) = Stack(0) + Stack(1)), pop	ADD $r1 = $r16 + $r17	
iload_3	move_int (local > Stack(0))		1
iload_4	move_int (local > Stack(0))	ADD $r2 = $r18 + $r19	
iadd	add_int(Stack(1) = Stack(0) + Stack(1)), pop		
isub	sub_int(Stack(1) = Stack(0) Stack(1)), pop	SUB $r1 = $r1 $r2	2
istore_5	move_int (Stack(0) > local), pop		
iload_5	move_int (local > Stack(0))		
ireturn	returnarg_int(Stack(0))	MOV $r16 = $r1	3

Fig. 3. Example of translation and scheduling of a sequence of Bytecodes.

long as possible a relation between a slot and a register, in this case between the deepest slot of the Java Frame and the register *$rx*. This mapping policy generates a *Write After Read* (WAR) dependency, between the third and the second instruction, that can be removed using the cyclic register allocation. The last column highlights how this technique is applied: when the deepest slot is rewritten, it is associated to the first available register, which will be different from *$rx*. In this case, since there are no data dependencies between the second and the third instruction, the scheduler can switch them.

However, the cyclic register allocation has one weak point, that is it may force the code generator to produce extra spills, as more registers are in use. This phenomenon has limited impact on the spills that are created at branches, since the values that need to be preserved are kept in global registers, but it has some impact when a call is produced that was not in the Java Bytecode control flow, as, for example, is the case for operations supported through library functions, such as the 64 bit integer and the floating point operations in the Lx processor. We will revisit this issue in Section 5.

4.2. *Instruction scheduling*

The scheduling algorithm is the critical part of any compiler for a VLIW architecture, since it conditions the exploitation of *instruction level parallelism*. In a dynamic compiler, there is a trade-off between the schedule quality and the scheduling time. Therefore, only fast instruction scheduling algorithm are deemed acceptable is this context.

There are two main classes of scheduling algorithms, those that work at basic block level and those that consider also control flow structures. We chose to implement a basic block scheduler for two main reasons: it is required for the development of more powerful schedulers, and it can guarantee low scheduling times. A survey of the different scheduling algorithms can be found in [21].

Our scheduler implements a greedy algorithm, based on the *operation scheduling* paradigm, and applying the *As Soon As Possible* (ASAP) strategy. The operation scheduling method chooses, at each iteration, an instruction from a ready set, and tries to insert it into the partial schedule, while preserving compatibility with data dependencies and resource constraints. In our case, the next available instruction is the last instruction generated by the JIT Translator and the scheduler tries to schedule it in the earliest possible position.

The scheduler scope is limited by a scheduling window, which guarantees an upper bound to time complexity, as the number of potential positions (or *slots*) to check for an instruction is limited by the scheduling window length. Within the window, the current instruction is first allocated to the last time slot, shifting the window if the instruction cannot fit. Then, the algorithm, described in Fig. 5, attempts to move the current instruction up in the window, stopping when a dependency is reached, and skipping time slots where the current instruction cannot be allocated due to lack of resources. Branch instructions are always scheduled last in the window, and the window is flushed when a basic block ends. The window size is a parameter of the algorithm. We tested the scheduler with window sizes ranging from 2 to 20. The optimum size for the benchmarks described in Section 5 was found to be 8.

Figure 6 explains how the scheduler works, showing a real case of instruction stream reordering, under data

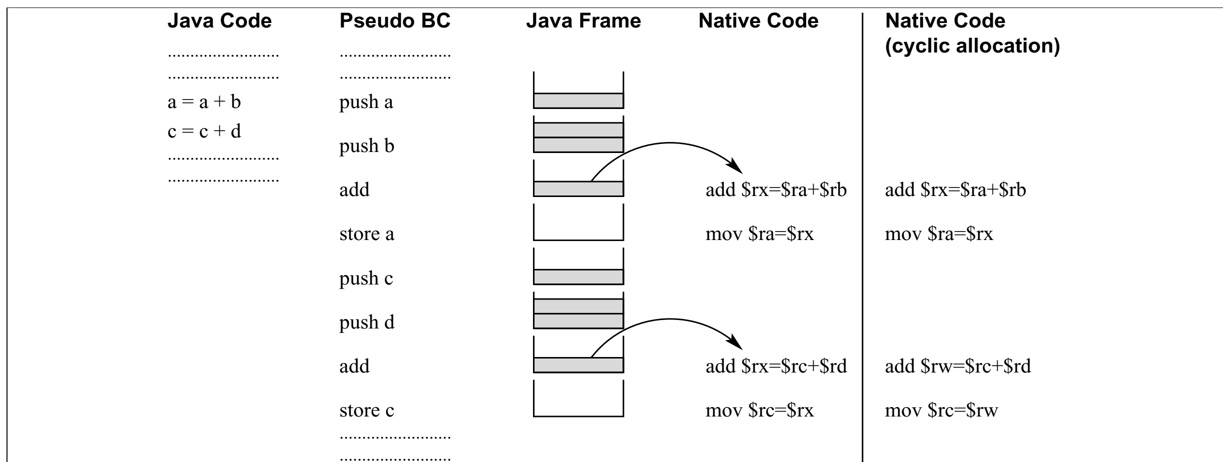

Fig. 4. Example of cyclic allocation politics.

```
void schedule(instr *cur_instr) {
  if (check_deps(cur_instr, index) ||
      arch_constraints(cur_instr, last(sched_win)))
      shift(sched_win, 1);
      sched_time = last(sched_win);
        if !is_branch(cur_instr)
    for (index = last(sched_win);
         index < first(sched_win);
         index = index - 1 ) {
             if (check_deps(cur_instr, index)) break;
             elsif (!arch_constraints(cur_instr, index))
                 sched_time = index;
    }
  emit_instr(cur_instr, sched_time);
}
```

Fig. 5. Basic instruction scheduling algorithm.

dependencies and resources constraints. We have considered a VLIW machine that can consume four ALUs and only one *Load Store Unit* (LSU) at each clock cycle. Moreover, in this example, we have limited the shifting window size at two.

At stage (`a`) the shifting window is empty and the scheduler can insert the incoming instruction, `mov $ra=0`, in the first *slot* of the first *bundle*. The next incoming instruction, `mov $rb=4`, is scheduled in parallel with the first one, filling the second slot of the same bundle: this is legal because there are no data conflicts between the two instructions (`b`). The third instruction of the stream, `add $rc,$ra,$rb`, which adds `$ra` and `$rb` and put the result in `$rc`, can be scheduled only in the second bundle, as shown in (`c`). This is due to a *Read After Write* (RAW) dependency between the current instruction, which reads registers `$ra` and `$rb`, and the previous instructions that write the same registers. For the same reason, the next incoming instruction, `mov $rx=$rc`, can be scheduled only at clock cycle three: since there is no room into the scheduling windows, it will be *shifted* (`d`); this means that, from now on, the scheduling algorithm cannot reach the first bundle – it will be impossible to use the free slots of that bundle. The last two instruction (`e,f`) could be scheduled in parallel, even though there is a *Write After*

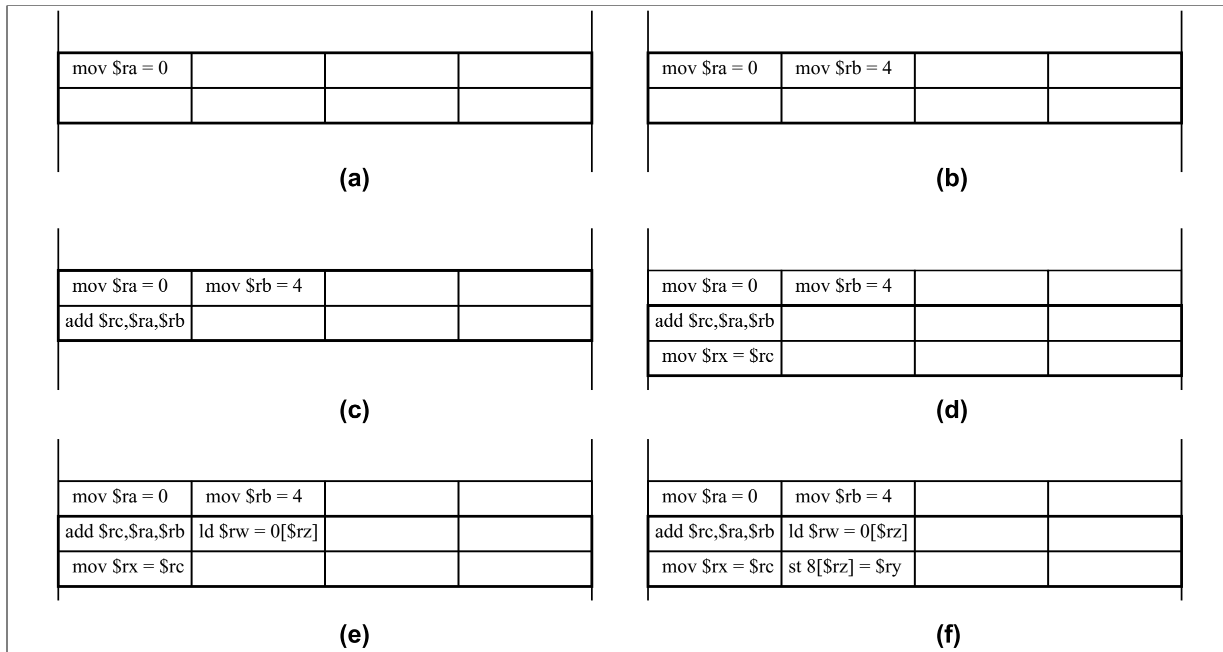

Fig. 6. Example of shifting windows and ASAP policy.

Read (WAR) data conflict between `st (store)` and `ld(load)`: the execution model imposes that the *write result* steps be executed, within the same clock cycle, after the *read operand* steps. However, we must schedule the sixth instruction after the fifth, because only one load/store unit is available per clock cycle.

The algorithm presented hereabove is able to capture much of the *ILP* available in the basic block: we tested it offline against a traditional list scheduler, and it did not show a significant degradation in performances. However, it has no ability to detect and exploit parallelism across different basic blocks.

A second scheduling option was therefore added, where the algorithm can move instruction across a branch boundary by predicating them. *Move* instructions are transformed into *select* instructions with the same predicate as the branch crossed.

Other instructions that write a register are split into two parts, as shown in Fig. 7, where the first part executes the operation, and writes to a temporary register, while the second part is a predicated move between the temporary and target register. In this schema, the scheduling window is not flushed at the end of a basic block if an opportunity for inter-basic block scheduling is detected.

Usually, only instructions that do not modify the control flow nor raise exceptions can be moved. However, instructions that raise exceptions can be move if they have a non-excepting equivalent, which is their used in their stead. This modifies the semantics of the program to a certain extent, so the global scheduler can be only applied if this change – i.e., not raising an exception when a load instruction fails – is acceptable.

4.3. Memory disambiguation

One of the major constraints in instruction scheduling is imposed by aliasing of memory location. Basically, the memory is considered as a single location for the purpose of deciding whether the scheduler can move an instruction without having it overwrite a datum needed by another operation, or read a datum that has not yet been written [17].

Since alias analysis based on flow is computationally intensive, we perform memory disambiguation on binary code. In this case the names of addresses are of the type $k[\$rx]$, where k is a constant, $\$rx$ is called *base register*, and the memory address is computed as k + *contentOf*(*$rx*).

Our *instruction inspection* technique works within a basic block, and aims at reaching two different goals:

- recognize names that are always alias and apply, if possible, a copy propagation pass;
- detect names that are not alias and then exploit the collected information to remove scheduling constraints between instructions using them.

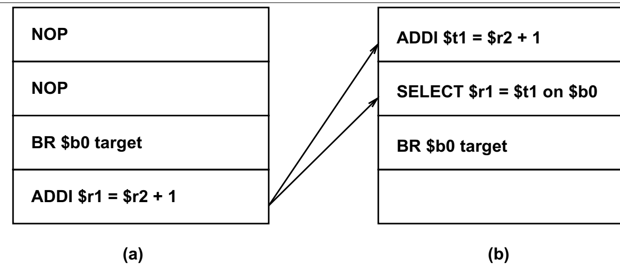

Fig. 7. Crossing a BB boundary.

As for the first goal, we exploit the interface provided by `jit3` of Kaffe to handle copy propagation: the analysis information that two slots of the *java stack* are aliases is passed to the Kaffe framework, so that the register allocator attempts to map them to the same register; if the attempt is successful an eventual copy is not translated to native instructions, thereby performing dead code elimination *on the fly*.

For the second goal, we collect information about names that are never alias, exploiting the following sufficient conditions for two memory locations to be non-conflicting:

1. they use distinct offset from the same base register: $k_1[\$rx_1] \neq k_2[\$rx_2] \Leftarrow x_1 = x_2 \wedge k_1 \neq k_2$;
2. one uses a register known to point to the stack and the other uses a register known to point to the global data area: $k_1[\$rx_1] \neq k_2[\$rx_2] \Leftarrow (\$rx_1 = sp \wedge \$rx_2 \neq sp) \vee (\$rx_2 = sp \wedge \$rx_1 \neq sp)$, where sp is the current stack pointer value;

Specifically, we use information on the store and load operations created by basic block prologues and epilogues, which always write and read values to and from stack locations that are never alias due to condition 1.

This form of local alias analysis gives the scheduler more freedom to move memory operations for out-of-order execution.

5. Experimental evaluation

This section presents the experimental results and their interpretation.

In order to understand the effect of scheduling and optimization on performances, we have run several benchmarks with a range of input data sizes, and we have extracted two quantities:

1. The asymptotic speedup obtained by the four `jist-`*i* versions versus unscheduled `jit` execution, an indicator of the maximum performance improvement obtained by optimization and runtime scheduling;
2. The break-even point of each `jist-`*i* and `jit`, defined as the minimum problem size allowing the compiler to provide an increase in performance that recovers the degradation caused by optimization and scheduling times.

A few words are needed to justify the set of benchmarks and the experimental setting. Benchmark suites for embedded Java application were not readily available, and in any case their significance would be subject to criticism because most embedded systems include special hardware devices and native libraries. Moreover it was impossible to run large Java applications on the Lx simulated machine with reduced Java library support available, therefore we decided to perform measurements on smaller programs or kernels, including both basic algorithms and operations relevant to embedded systems.

We also decided to cut the lengthy start-up time of the VM, from our measurements, by running the benchmarks over a lightweight, reduced version of Kaffe JIT, called JitBasic, which allows the execution of almost any Java method with the exclusion of I/O primitives. Including start up times would have distorted our data by measurements of code not generated by JIST. However a few runs were also performed with the full Java VM, to compare the performance of the interpreter and the other execution engines.

We have selected a mix of artificial benchmarks and real world application kernels. For example, some of our routines implement numerical methods usually performed in image and digital signal processing. Image processing is representative of typical applications of embedded systems.

The benefits highlighted by the results can be reproduced on methods having the same computational complexity and working on the same type of data. We also expect that our results indicate a practical limit, in

Table 1
Maximum size of parameters

Benchmark	Parameters	Max Size
`Matrix`	`N`, the matrix order	$N = 120$
`Cholesky`	`S`, the number of iterations; `N`, the matrix order	$S = 100$, $N = 1000$
`LZcompr`	`N`, length of the array	$N = 4000$
`Mean`	`N`, the matrix order; `S`, the number of iterations	$N = 300$, $S = 3$
`Gauss`	`N`, the matrix order; `S`, the number of iterations	$N = 300$, $S = 3$
`Dijkstra`	`N`, the number of vertices of the graph; `S`, the number of iterations	$N = 1000$, $S = 4$
`Sieve`	`N`, interval's right bound; `S`, the number of iterations	$N = 40000$, $S = 100$
`BubbleSort`	`N`, length of the array	$N = 4000$

term of performance improvements, valid for all VLIW architectures comparable with Lx.

The benchmark set is composed of the following routines:

- Integer matrix multiplication algorithm (`Matrix`);
- Cholesky's floating point matrix factorization algorithm (`Cholesky`);
- LZ compression algorithm on an array of bytes (`LZcompr`);
- Mean filter algorithm (`Mean`);
- Gaussian filter algorithm (`Gauss`);
- Dijkstra algorithm to compute minimum distance between two vertices in a direct graph (`Dijkstra`);
- Sieve of Eratosthenes algorithm (`Sieve`);
- BubbleSort algorithm on an array of bytes (`BubbleSort`);

All the benchmarks are parametric with respect to either the size of the problem, the number of iterations, or both. Table 1 lists the parameters for each benchmark, and shows the maximum value of the parameters used in the experiments.

A general issue to be discussed is how the arithmetic for floating point and 64-bit integer is implemented. For the intended applications of the Lx processor, floating point computation is either not needed or provided through a coprocessor, which is not available in the simulator. For 64-bit integers, the current implementation of JIST uses emulation via software calls for multiplication and division. For this reason, the benefits are limited for benchmarks working on floating point data, like `Cholesky`. The introduction of a Floating Point Unit should make the results comparable with what obtained for integer data.

The reported execution times (ms) refer to an Lx processor working at 250 MHz.

5.1. Interpretation vs. JIT compilation

The first set of results concern the performances of the basic JIT compiler (`jit`) with respect to the interpreter (`intrp`). The interpreter benefits from compile-time scheduling, as all parts of the VM written in C are compiled with a production – level compiler targeted to the Lx. On the other hand, code generated by the basic version of the JIT compiler is not scheduled, which gives an additional edge to the interpreter.

Figure 8 reports the performances of the `BubbleSort` benchmark on the full Java VM. Except for small values of the parameter, the performances of the interpreter are so much slower that it quickly became impossible to gather experimental data. Data for the other benchmarks are consistent with those presented here and with previous literature [?], with the performance gap between `intrp` and `jit` fast growing with the problem size. The interpreter is therefore an efficient solution only for short running benchmarks (less than 1 s in the `BubbleSort` case), where the total execution time is dominated by the cost of the virtual machine startup.

5.2. JIT optimizations

The next results concern the speedup obtained on baseline (`jit`) with four different versions of `jist`, all running on JitBasic, to separately analyze the impact of different optimization. As shown in Table 2 the first (`jist-1`) only adds basic block instruction scheduling, the second (`jist-2`) adds both scheduling and optimized register allocation, and the third (`jist-3`) adds inter-basic block scheduling as described in Section 4.2, coupled with a different policy of window flushing. The last optimization performed is memory alias disambiguation (`jist-4`), which allows the scheduler to remove several false dependencies between couples of memory operations (loads and stores).

Figures 9 and 10 show the performance and speedup for the `BubbleSort` benchmark. As the number of iterations grows, the performance gap between the basic `jit` and the optimized versions increases. The speedup chart shows that, in this case, the scheduled

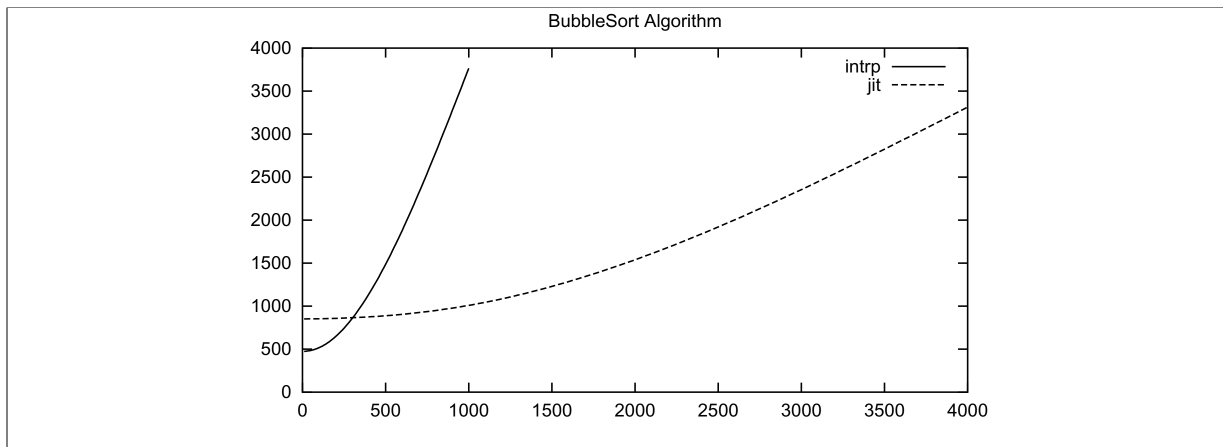

Fig. 8. Execution time for BubbleSort (ms).

Table 2
`jist` and `jit` configurations

	Instruction Scheduling	Register Allocation	Memory Disambiguation
`jit`	none	basic	no
`jist-1`	intra-BB	basic	no
`jist-2`	intra-BB	optimized	no
`jist-3`	inter-BB	optimized	no
`jist-4`	inter-BB	optimized	yes

code is slower than the unscheduled one only for small problem sizes. The impact of register allocation is also highlighted, as the two `jist` version that benefit from optimized register allocation achieve a maximum speedup higher than 20%, while the standard Kaffe register allocation cannot reach 15%. The `jist-3` JIT performs always better than the `jist-2` engine, achieving near 30% speedup over `jist-1`.

Figure 11 summarizes the maximum speedups achieved during our tests, corresponding to the highest values of the parameters, specified in Table 1. The corresponding simulated runtimes (ms) are reported in Table 3.

Notice that the results are consistent with those shown in the `BubbleSort` case (Fig. 10), except for the `jist-3` JIT which is always able to compensate its overhead, but the benefits reached are not always significant. This is due to the fact that, currently, the `jist-3` works on rather small regions of code. However, the results achieved on benchmarks like `Bubble` and `LZcompr` lead us to believe that method inlining and a more aggressive inter-basic block scheduling policy could make the `jist-3` JIT beneficial on most benchmarks.

In order to further evaluate the impact of register allocation, we have traced the data dependency constraints which were detected during scheduling. We noticed that the *cyclic register allocation* reduces the data dependencies among registers; in the `Matrix` benchmark, for example, the false dependencies are reduced by 81%.

Table 5 provides a report of the data dependency constraints that were detected during scheduling of `Matrix`. The attempts to write in register that needs to be preserved (*WREG*) are false dependencies and are addressed by the cyclic register allocation policy. The other classes represent constraints on condition registers (*WBREG* and *RBREG*) and memory (*WMEM* and *RMEM*), and true dependencies on registers *RREG*. The register allocation is able to reduce the number of false dependencies on registers (*WREG*).

Some considerations are necessary about the *Break Even Point* (BEP). This quantity is the size of the problem for which the costs of scheduling and optimizations are balanced. Table 4 shows the BEPs between `jist-4` and `jit` executions. For these experiments, we performed a single iteration of each method (i.e., we set the `S` parameter to 1, where applicable), varying only the size of input data. We can notice that the obtained values are reasonably small; for example, Lempel-Ziv compression reaches its BEP for data smaller than 1 KB.

Concerning parallelization, one should keep in mind that the average number of instructions issued at each clock cycle (*IPC*) for C code performing the same operations as in our benchmarks ranges between 1.6 and 1.9 depending on the benchmark. Table 6 reports the IPC for the longest runs of each benchmark program. The IPC shown include two delays: explicit latencies, and therefore cycle spent performing no-operation instructions (*nop*); and processor stalls due to cache misses and mispredicted branches, which explains the low figures.

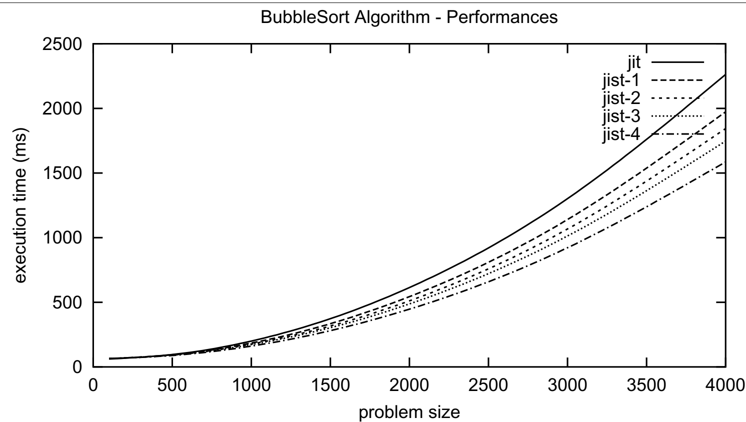

Fig. 9. Performance of the BubbleSort benchmark using `jit` as comparison.

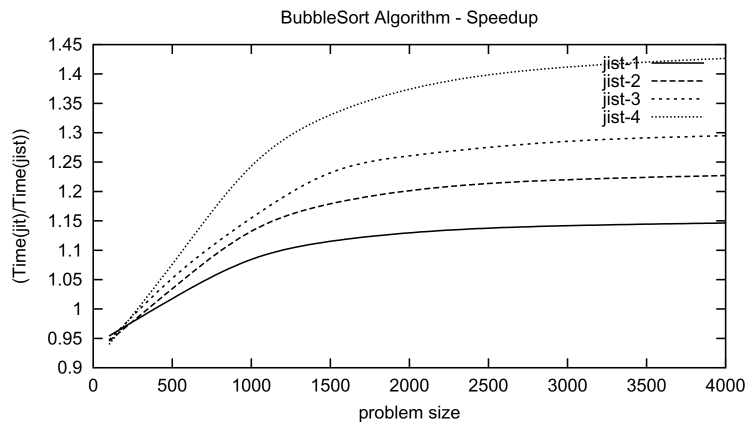

Fig. 10. Speedup of the BubbleSort benchmark using `jit` as comparison.

Table 3
Benchmarks performances for the highest values of the parameters (ms)

Benchmark	`jit`	`jist-1`	`jist-2`	`jist-3`	`jist-4`
Matrix	2087.55	1707.96	1551.45	1551.31	1422.68
Cholesky	563.08	543.83	527.78	524.62	521.90
LZcompr	1438.13	1309.43	1311.87	1242.43	1085.12
Mean	811.66	706.51	649.22	643.09	586.85
Gauss	892.51	732.46	681.38	674.21	608.63
Dijkstra	2421.39	2148.19	2036.43	2005.64	1859.09
Sieve	1895.95	1625.40	1495.99	1488.68	1489.24
BubbleSort	2262.80	1973.85	1843.89	1747.32	1586.12

We can notice that `Cholesky` is the only benchmark with IPC always greater than one. The phenomenon can be explained saying that `Cholesky` works on floating point data, and the floating point operations are simulated via software call to functions which have a good degree of parallelism, since they are statically compiled.

Table 7 shows the frequencies of each instruction class (Memory, Arithmetic-Logic and Branch) in the benchmark programs. These figures are measured on the longest benchmark runs, in order to reduce the impact of the JIT compiler code in the instruction statistics. The ALU entry represents both the standard ALUs and the multipliers, since multiplication are infrequent, between 1% and 3% depending on the benchmark. Data for the Cholesky benchmark is biased by the large

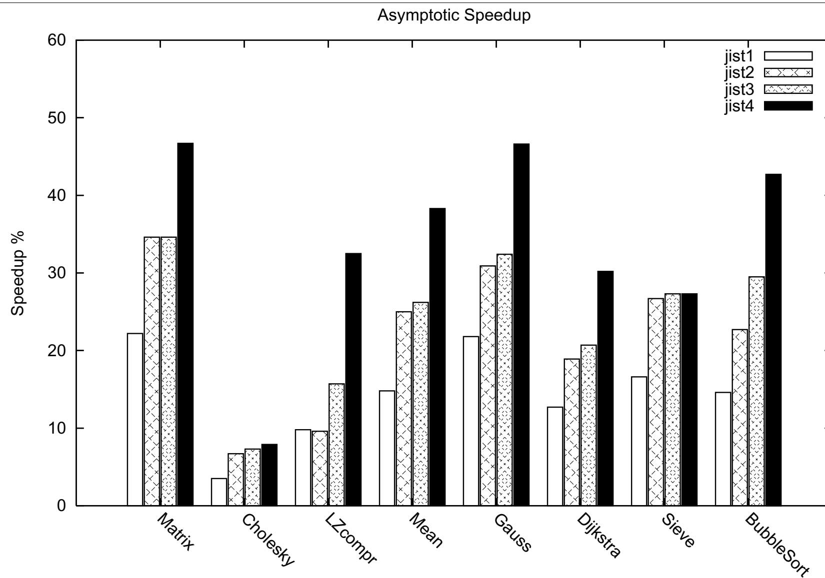

Fig. 11. Maximum speedups achieved.

Table 4
Single run BEPs between `jist-4` and `jit`

Benchmark	BEP
Matrix	$N = 47$
Cholesky	$N = 350$
LZcompr	$N = 1010$
Mean	$N = 140$
Gauss	$N = 132$
Dijkstra	$N = 250$
Sieve	$N = 38000$
BubbleSort	$N = 370$

Table 5
Data dependencies

Matrix	Kaffe RegAlloc	Cyclic RegAlloc
WREG	73	9
RREG	179	166
RBREG	24	24
WBREG	0	0
WMEM	38	41
RMEM	0	9

use of floating point operations, and therefore statically compiled native methods.

The data clearly show a dominance of memory instruction, which becomes important if these frequencies are weighed on the number of functional units able to serve them. The Lx Processor simulated in the experiments has a single memory unit and four ALUs. In spite of the introduction in `jist-4` of memory disambiguation, the load/store unit remains the bottleneck of the system. One way to overcome it, other than by extending the hardware configuration, could be to employ more powerful alias analysis, in order to solve false dependencies on memory. However, this solution costs an additional alias analysis pass and its effectiveness remains to be measured.

The scheduling overhead of `jist-4` (all optimizations added) is about 0.5% of the runtimes of Table 3, a rather negligible fraction, even though it is 25% of the overall compilation cost.

To give a metric of cost efficiency of the implemented optimizations, we propose a normalized cost/benefit ratio. Let $O_{i,p}$ be the overhead of an optimization pass `jist-`*i* on benchmark program *p*, $P_{i,p}(N)$ be the execution time depending on problem size *N*, $O_{0,p}$ be the compilation overhead of `jit`, and $P_{0,p}(N)$ be the execution time on `jit`. Let $S_{i,p}(N) = P_{0,p}(N)/P_{i,p}(N)$ be the speedup with respect to `jit` and $C_{i,p} = O_{0,p}/O_{i,p}$ the slowdown due to the optimization cost. Let $\overline{S}_{i,p}$ be the value for $max_N(S_{i,p}(N))$. Then $R_{i,p} = C_{i,p}/\overline{S}_{i,p}$ is the cost/benefit ratio for `jist-`*i* on program *p*. Since $R_{i,p}$ is normalized with respect to cost and benefit for `jit` on program *p*, we can obtain an average R_i, which ranges from $R_1 = 0.95$ to $R_4 = 0.86$ (standard deviation are always less than 0.08). In conclusion the increase in cost of the various `jist-`*i* is always smaller than the increase in performances they can provide. It remains for future investigation to discover where the law of diminishing returns starts to operate for more aggressive optimizations.

Table 6
Instructions issued per Clock Cycle

Benchmark	jit	jist-1	jist-2	jist-3	jist-4
Matrix	0.54	0.67	0.74	0.74	0.80
Cholesky	1.08	1.13	1.17	1.18	1.19
LZcompr	0.52	0.58	0.58	0.61	0.70
Mean	0.75	0.90	0.99	0.99	1.09
Gauss	0.75	0.95	1.04	1.04	1.15
Dijkstra	0.66	0.75	0.79	0.81	0.88
Sieve	0.59	0.67	0.72	0.78	0.73
BubbleSort	0.52	0.60	0.64	0.67	0.74

Table 7
Instruction class frequencies

Benchmark	MEM	ALU	BRU
Sieve	46%	40%	14%
Matrix	50%	42%	8%
LZcompr	48%	37%	15%
Mean	42%	49%	9%
Gauss	40%	51%	9%
Cholesky	28%	59%	13%
BubbleSort	46%	42%	12%

5.3. *Comparison with related approaches*

There are several different ways to optimize Java programs: in addition to JIT compilers, *Ahead-Of-Time* (AOT) compilers, Java processors and native libraries are all used to this end.

AOT compilers [23] move the compilation of the Java bytecode ahead of the start of the execution. While this can provide a degree of reduction of the compilation overhead, the benefit can only be obtained if the program can be compiled before it is executed – that is, if the need to execute the program is detected early, the program is fully available (there is no need for dynamically loaded libraries), and sufficient space is available in memory to hold the compiled program. If these requirements are not met, AOT compilation is no better than JIT compilation. Our results are therefore useful even in AOT settings.

Java processors [24] are specific pieces of hardware that execute Java bytecode. They may provide large improvements in performances, but cannot take advantage of the large amounts of redundancy in Java code, and are not normally able to exploit parallelism. Moreover, Java processors only execute subsets of the Java bytecode language.

Native libraries, interfacing with Java programs through the Java Native Interface (JNI) are a viable, efficient way to deal with performance requirements in Java programs. However, the use of native libraries directly conflicts with one of the strongest points for the adoption of Java: native libraries are not portable, since they are written and compiled for a specific architecture.

6. Conclusions and future work

We have described JIST, a Java bytecode compiler targeted to the HP/STM Lx processor, a small VLIW chip used for embedded systems. This research project was planned in order to assess the performances of Java programs on such inexpensive processors. Effective instruction scheduling and register allocation approaches have been chosen, and their performances systematically evaluated for a set of small but significant benchmarks. Due to the choice of the Kaffe JIT compiler, the scheduler operates on basic blocks. As a first step towards more global scheduling, we have also studied code movement across blocks. Another step towards more aggressive optimization is the introduction of a simple form of memory disambiguation, which allows a more effective use of the single load/store unit. The cost effectiveness of the optimizations has been measured both one by one and collectively. The overall picture fills a gap in the available literature on dynamic compilation and confirms the validity of bytecode JIT translation for simple VLIW targets.

Through the optimizations implemented in JIST, scheduled code can achieve maximum speedups ranging from 20% to 45% in terms of the execution time with respect to sequential code. These results, though obtained by dynamic transformations, are comparable with those achieved by static compilers that do not exploit parallelism between different iterations of a same loop: a recent survey [21] reports that the performance improvement achieved on multiple instruction issue processors by statically scheduling basic blocks may range from 20 to 50%. Moreover, our code generator and scheduler can achieve the performance described in [6], in the ideal case assumed therein. More improvements should be obtainable through more aggressive transformations.

Currently, we are building two extensions to our Virtual Machine and JIT compiler:

– A global scheduling framework, aiming at the implementation of a scheduling algorithm akin to Trace Scheduling [25] or Superblock Scheduling [26];
– A selective compilation and optimization framework, to enable the VM to select between compiled and interpreted execution of each method, and to activate optimizations only on the kernel regions of a method, based on information from profiling and compiler hints (from the Java to bytecode compiler).

Future research will also experiment additional global scheduling algorithms, and other optimizations such as method inlining. The final goal is to discover the border line where the optimization costs outweigh the benefits for typical embedded application benchmarks. Another direction for future works involves evaluating the combination of VLIW architecture and dynamic compilation of portable bytecode, as the building blocks of non-homogeneous distributed systems. This direction would combine the effectiveness of VLIW architectures in exploiting instruction – level parallelism with the capabilities of dynamic compilation to provide a virtualized view of the architecture, allowing for a degree of object-code compatibility not normally available in VLIW.

Acknowledgments

The authors wish to acknowledge Giuseppe Desoli, Marco Garatti and Erven Rohou of STMicroelectronics labs in Manno (CH) for their support.

References

[1] Ø. Strøm, K. Svarstad and E.J. Aas, On the Utilization of Java Technology in Embedded Systems, *Design Automation for Embedded Systems* **1** (2003), 87–106.
[2] A. Corsaro and D.C. Schmidt, *Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems,* In RTAS '02: Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02), IEEE Computer Society, 2002, 20.
[3] B. Blount and S. Chatterjee, An Evaluation of Java for Numerical Computing, *Scientific Programming* **7**(2) (1999), 97–110.
[4] J.E. Moreira, S.P. Midkiff, M. Gupta, P.V. Artigas, P. Wu and G. Almasi, The NINJA Project, *Commun. ACM* **44**(10) (2001), 102–109.
[5] G.C. Fox and W. Furmanski, Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modeling, *Concurrency: Practice and Experience* **9**(6) (June 1997), 415–425.
[6] K. Ebcioglu, E. Altman and E. Hokenek, *A Java ILP Machine Based on Fast Dynamic Compilation,* In MASCOTS'97 - International Workshop on Security and Efficiency Aspects of Java, 1997.
[7] M. Tremblay, *MAJC-5200: A VLIW Convergent MPSOC,* In Microprocessor Forum '99, 1999.
[8] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi and J. Fisher, *DELI: a New Run-time Control Point*, In 35th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-35), 2002.
[9] K. Gough, *Stacking Them Up: A Comparison of Virtual Machines,* In Australasian Computer Systems Architecture Conference, Goldcoast, Queensland Australia, IEEE Computer Society Press, 2000, 52–59.
[10] J. Aycock, A Brief History of Just-In-Time, *ACM Computing Surveys* **35**(2) (June 2003), 97–113.
[11] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y.C. Chung, S. Kim, K. Ebcioglu and E.R. Altman, *LaTTe: A Java VM Just-In-Time Compiler with Fast and Efficient Register Allocation,* In IEEE PACT, 1999, 128–138.
[12] M. Chen and K. Olukotun, *Targeting Dynamic Compilation for Embedded Environments,* In JVM'02, August 2002.
[13] K. Ebcioglu and E.R. Altman, *DAISY: Dynamic Compilation for 100% Architectural Compatibility,* In 24th International Symposium on Computer Architecture (ISCA), June 1997, 26–37.
[14] T.M. Conte and S.W. Sathaye, Optimization of VLIW Compatibility Systems Employing Dynamic Rescheduling, *International Journal of Parallel Programming* **25**(2) (1997), 83–112.
[15] B.R. Rau, *Dynamically Scheduled VLIW Processors,* In 26th Annual Intl. Symp. on Microarchitecture, Dec 1993, 80–92.
[16] E. Duesterwald, Dynamic Compilation, in: *The Compiler Design Handbook – Optimizations and Machine Code Generation,* Y.N. Srikant and Priti Shankar, eds, CRC Press, 2003, pp. 739–761.
[17] S.K. Debray, R. Muth and M. Weippert, *Alias Analysis of Executable Code,* In Symposium on Principles of Programming Languages, 1998, 12–24.
[18] P. Faraboschi, G. Brown, J.A. Fisher, G. Desoli and F. Homewood, *Lx: a Technology Platform for Customizable VLIW Embedded Processing,* In The 27th Annual International Symposium on Computer architecture 2000, ACM Press, New York, NY, USA, 2000, 203–213.
[19] Kaffe, http://www.kaffe.org/.
[20] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification,* Reading, Addison-Wesley, MA, 1997.
[21] R. Govindarajan, Instruction Scheduling, in: *The Compiler Design Handbook – Optimizations and Machine Code Generation,* Y.N. Srikant and P. Shankar, eds, CRC Press, 2003, 631–687.
[22] I.H. Kazi, H.H. Chen, B. Stanley and D.J. Lilja, Techniques for Obtaining High Performance in Java Programs, *ACM Computing Surveys* **32**(3) (2000), 213–240.
[23] T.A. Proebsting, G. Townsend, P. Bridges, J.H. Hartman, T. Newsham and S.A. Watterson, *Toba: Java For Applications, A Way Ahead of Time* (*WAT*) *Compiler,* In Proceedings of the Third Conference on Object-Oriented Technologies and Systems, June 1997.

[24] H. McGhan and M. O'Connor, PicoJava: a Direct Execution Engine for Java Bytecode, *IEEE Computer* **31**(10 (Oct 1998), 22–30.
[25] J.A. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. on Computers* **30**(7) (July 1981), 478–490.
[26] W.-M.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm and D.M. Lavery, The Superblock: an Effective Technique for VLIW and Superscalar Compilation, *J. Supercomput.* **7**(1–2) (1993), 229–248.