The CoGenT Project: Co-Generating Compilers and Simulators For Dynamically Compiled Languages

J. Eliot B. Moss, Charles C. Weems, and Timothy Richards
Department of Computer Science, University of Massachusetts Amherst

Abstract

To understand the performance of modern Java systems one must observe execution in the context of specific architectures. It is also important that we make these observations using a compiler that is capable of producing optimizations that are specialized to the target machine. Current architectural simulators, however, provide little or no support for dynamically compiled languages and their environments. At the same time, high-quality optimizing compilers lack the flexibility needed to be quickly retargeted to architectural variations.

Experimenting with innovative architectural ideas requires modifications not only to the simulator, but to the compiler backend as well. Currently, it is difficult if not impossible to accomplish exploration of this sort.

This paper proposes a system for the coordinated effort of generating simulators and matching compiler backends automatically from machine descriptions. Machine descriptions are processed by tools to produce efficient compiler and simulator components and these components "plug in" to an existing framework. This system provides an exploratory environment for compiler writers, computer architects, and students, while maintaining the performance and flexibility required for evaluating real systems.

1. Introduction

This project is motivated by a substantial obstacle we have encountered in trying to explore, in detail, the performance of Java virtual machines on modern hardware, and to evaluate architectural features that might improve performance. The problem is four-fold. First, to consider performance questions accurately and in detail requires a cycle-accurate simulator, and it must be capable of supporting the dynamic environment of a modern Java system. Second, realistic performance predictions require compiler optimizations, such as instruction scheduling, tuned to the architecture. Third, we must be able to generate new compiler optimizers (back-ends) with matching simulators. Fourth, the generation process, and the generated simulators, also need to be relatively efficient, so as to produce results in reasonable time. The solution is to produce compiler and simulator components automatically from machine descriptions, in the context of a flexible framework that joins the generated components in a highly efficient manner.

2. The Need

One way to grasp the need is to consider the prototype system illustrated in Figure 1 (some pages later in this proposal). It shows *ten* components that will be generated from descriptions. These components *must* be built in order to target the compiler and simulator to a specific architecture and implementation, and obtain the desired measurements

We are not the only ones arguing for this kind of automation. Schnarr built the Facile simulator generator [23, 21] because his simulators were too complex to build reliably by hand. Ramsey and Fernández [17, 19] and Ramsey and Davidson [18] emphasize the need for machine instruction specifications and tools and generators that work from them. Bailey and Davidson describe the ambiguities and errors of calling conventions described in English [4, 3]. The ML-RISC project [9] also places automatic generation of compiler components as a priority.

We believe that enough pieces of the overall problem have been addressed that it is reasonable, though still challenging, to produce coordinated compiler and simulator components automatically. We next review prior work, and proceed to lay out our vision in more detail.

3. Prior Work

Many parts of this overall problem have been addressed previously, but the overall combination has not:

- Most cycle accurate simulators demand statically linked executables, and so cannot handle Java, which generates code dynamically. They also tend not to handle hardware traps, some of which are reflected to

0-7695-1926-1/03/$17.00 (C) 2003 IEEE

the application as Java exceptions. We extended SimpleScalar [6] to produce Dynamic SimpleScalar [10], which took more than a year to build and is not easily retargeted.

- Full system simulators such as SimOS [20], which simulate the operating system and hardware devices, get around those problems, but most of these are functional simulators only and do not provide timing information.
- Most easily retargeted simulators are not cycle-accurate, but model only functional semantics. Some offer cache hit-miss or branch prediction statistics models.
- There is considerable previous work on retargetable code generators, but most of them require one to present code generator patterns or rules, such as those of Twig [1] or burg [8]. We want to generate the code generator (i.e., instruction selector) directly from an ISA description. This is somewhat similar to Cattell's dissertation work [5], which has apparently not been followed up. However, we also want to derive the costs in the rules automatically from machine timing descriptions.
- Not only do we need components to be readily retargeted, they need to be, as much as possible, generated automatically from common descriptions. One reason is that we need to make sure that the compiler and simulator match each other. Another reason is that automation will lead to fewer mistakes and also to more rapid construction of compiler-simulator pairs.
- The generated simulators need to be reasonably efficient. Previous techniques to speed simulation include predecoding and caching (common techniques), state-transition caching [21, 22], using threaded code [13], and, for essentially fixed targets, generating machine code at run time [16]. We intend to apply adaptive compilation, analogous to adaptive Java optimizing compilers, to speed simulation on-the-fly.

4. Our Vision

We now lay out our vision, treating in turn the generation of compiler components, the generation of simulator components, and efficient simulation.

4.1. Generating Compiler Components

We consider three target-specific tasks in generating optimized code: instruction selection (code generation), register allocation, and instruction scheduling.

Instruction selection: Of the three tasks, instruction selection is conceptually the most difficult. Up to a certain point in a compiler, the processing and optimization is more or less independent of the target architecture, but at some point we must generate target instructions. Most techniques, such as BURS [8], work from some kind of machine-independent expression trees (e.g., register transfer language (RTL)). One writes a collection of target-dependent patterns. Each such pattern matches a bit of tree, reduces it to a smaller tree (usually a single node), and generates some target code in the process. The patterns include a cost. Tools have been available for some time that generate a pattern-matching tree-reduction (target) code generator given the patterns (usually called *rules*). This pattern matching technology is used in our Java system (the Jikes RVM, open source code produced by IBM Research [2]). Here are some examples of rules, writing the trees as LISP-like expressions:

Tree to match		Reduced tree	Cost	Code generated
const16	⇒	reg	1	li reg←const16
const32	⇒	reg	2	lui reg←hi(const32) ori reg←lo(const32)
(contents mem)	⇒	reg	2	lw reg←mem
(add reg$_1$ reg$_2$)	⇒	reg$_3$	1	add reg$_3$←reg$_1$,reg$_2$
(add reg$_1$ const16)	⇒	reg$_2$	1	addi reg$_2$←reg$_1$,const16
(assign mem reg)	⇒	reg$_1$	2	sw reg$_1$→mem

These rules are for the MIPS R2000, in which one can load or add an immediate 16-bit constant, but getting a 32-bit constant into a registers requires two instructions (a load-upper-immediate and an or-immediate). (One could also load from a global area, but it is not faster, and will be slower if there is a cache miss, etc.) For the input tree `(assign m1 (add (contents m2) c1))`, from a BURS-style code generator using the rules above we would obtain this minimal cost code:

```
lw r1←m2
addi r2←r1,c1
sw r2→m1
```

Such code generators target a machine with an infinite set of registers to hold temporary results; one applies register allocation later, adding any necessary spill stores and loads.

Retargeting such code generators is easy in the sense that one need only write a rule set for each processor. However, in our case we want to generate *the rules* from a description of the semantics of the ISA. This is more or less the problem that Cattell studied [5], and is also similar to the MLRISC back-end strategy [9]. (There are other such systems, but these are representative.) Fortunately, ISA semantics are written in a form in which it is not too difficult to extract many of the rules. In particular, we can generate a rule for every addressing mode and instruction form. This is not quite enough, though, as the `lui/ori` rule above suggests: to cover some intermediate language constructs may require multiple instructions. Discovering such sequences requires a certain amount of search, which is exemplified by Cattell's approach and also by *super-optimization* [14]. In this search one also applies algebraic rules, so that one can produce patterns such as these:

Tree to match		Reduced tree	Cost	Code generated
`(add const16 reg`$_1$`)`	⇒	`reg`$_2$	1	`addi reg`$_2$`←reg`$_1$`,const16`
`(sub const16 reg`$_1$`)`	⇒	`reg`$_2$	2	`subi reg`$_2$`←reg`$_1$`,const16` `sub reg`$_2$`←r0,reg`$_2$` // negate reg`$_2$

In generating rules, we need to make sure that every semantic tree can be *covered* (reduced), which implies that we can generate code for any tree. (If a particular intermediate representation (IR) cannot generate every possible tree, then we can relax this restriction to one that we cover every possible IR tree.) Once we have done that, we can apply super-optimization to each pattern's code sequence, to try to improve it. This requires a cost model, which we will derive from instruction timings (see below). Note that whereas BURS-style code generator generation is rather like parser generation in that it is concerned only with syntactic forms, we will be working in the realm of semantics, within the theories of integer and floating point arithmetic. Cattell took care, and so will we, not to slide down the slope into general theorem proving. Still, the search for an instruction sequence with given semantics is at heart an attempt to instantiate a theorem that there exists such a sequence. In practice, the search must apply heuristics and occasionally use judicious "advice" (in the form of theorems or transformation rules added by a human). Fortunately the need for such advice appears rare.

Register allocation: This is probably the easiest of the three components to deal with, because ultimately it relies primarily on tabular information about which registers are available on the target architecture, and any restrictions and conventions as to their uses. That is, we assume an essentially table-driven register allocator, and what we have to generate is the table. If the compiler framework is not table-driven, then it will require more effort to adapt the framework to automated use. This is probably an area where we could end up applying inordinate effort if we try to handle every possibility, so in practice our aims will likely be more modest. Still, the Jikes RVM handles register allocation for the Pentium (few registers) and the PowerPC (32 general purpose and 32 floating point registers), so the necessary algorithms are there for a useful range of possibilities. We note that MLRISC also includes a register allocation strategy, which we might exploit.

Instruction scheduling: This is clearly dependent not only on the target ISA, but also on the target implementation (timing). There are two popular approaches to instruction scheduling. One is to use resource vectors for each instruction. These indicate which resources (functional units, etc.) each instruction needs for each cycle of its execution. One tries to order the instructions so as to minimizes gaps in the schedules while obeying the resource constraints. Ways this to do this include heuristics and integer programming.

The other strategy is to use more local heuristics and build a schedule up one instruction at a time (list scheduling). List scheduling is fast and generally produces pretty good schedules. A commonly used heuristic is called *critical path* scheduling. Critical path scheduling requires a simple, cycle-level, machine model. We propose to generate that component automatically from the ISA and timing descriptions. In essence, it is a stripped down simulation component. We note that it would not be difficult (in principle at least) to generate resource vector information for other schedulers if that were desirable.

Compiler Framework: It should be clear that we will not be generating complete compilers from ISA and timing specifications. Rather, we will generate specific, focused, components that "plug in" to a substantially larger compiler designed to interface with them, in this case the existing Jikes RVM compilers.

4.2. Generating Simulator Components

We view a simulator as consisting of many kinds of components. Of these, the ones we propose to generate automatically from ISA and timing descriptions are instruction semantics and instruction timing, which we describe in more detail in Section 4.3. We will also generate the machine-dependent parts of debugger support. We assume that the remaining semantics and timing components present suitable interfaces to call and be called by the generated instruction-related components. We do plan to offer a modest range of options in a library of semantics and timing components, such as branch predictors, caches, and memory organizations.

Instrumentation is an area where we believe *aspect-oriented programming* [7] can be used to good effect. In this approach one specifies in a separate file the places where one desires to attach instrumentation, and what instrumentation code should be invoked at each place. This separate file is an *aspect*, in this case for instrumentation. The point is that even if instrumentation attaches at many places throughout the simulator code, the instrumentation remains specified separately. The aspect-oriented tool, e.g., AspectJ [11, 12], weaves together the aspect and the rest of the code.

4.3. Building Efficient Simulators

There are three areas of simulator efficiency we wish to discuss here, being most relevant: functional simulation of instructions, instruction timing simulation, and efficiency of the additional simulation components.

Functional simulation: The obvious and often used method of functional simulation somewhat literally models the hardware and its actions—except the hardware can be a simple conceptual model as opposed to real hardware with its internal parallelism and design for speed. One models the registers and memory as data structures in the programming language of the simulator, and writes a fetch, decode, execute loop. This loop uses the current program counter to index memory and fetch the bytes of the next instruction to execute. Then it does a case analysis on the bits of the instruction to determine what instruction was fetched, and uses some form of case statement (or similar dispatch structure) to branch to simulator code implementing that instruction's effects. The instruction's simulation code will need to fetch operands from the modeled registers and memory, perform computation on them, store results back to registers and memory, and update the program counter.

We offered this detailed description of a simulation instruction interpretation loop to give a sense of why such simulation might be slow. Here are some ways one can speed up functional simulation:

- *Pre-decode* the instructions.
- Use *threaded code* in the decoded instructions of the interpreter, reducing the per-instruction overhead.
- Take into account *more than the opcode* of the instructions.
- Generate code *at run time* that simulates the effect of a particular instruction.
- Generate code fragments for *more than one* instruction at once.
- *Optimize the code fragments* for more than one instruction.
- *Shift adaptively from slower to faster techniques* according to the execution frequency of instructions.

With the possible exception of adaptive run-time compilation, all of these techniques have been implemented before. However, to our knowledge no one has employed run-time code generation in simulators derived from ISA descriptions. We will explore adaptive run-time code generation of functional semantics from ISA descriptions. Here are some of the questions we will address:

- What are the challenges in producing good code quickly from ISA descriptions? Are some styles of ISA description better suited to this task than others?
- What are the relative speeds of interpretive, threaded-code, and native code forms with different levels of optimization? What are the relative code generation costs? What are the space-time trade-offs?
- What are good adaptive optimization triggers and trigger levels?

Generating functional simulation components: We can generate functional simulation components from ISA semantic descriptions by building what amounts to variant code generators. A normal code generator for the target matches target semantic trees against the IR and produces target instructions. In this case, we build IR for trees we want to simulate, and then generate *host* code for those trees.

For an instruction interpreter, we generate code, probably in a higher-level language such as Java or C, but possibly directly as host instructions, for a logical machine whose registers and memory are in simulator data structures. We compile this code as we build the simulator. Generating higher-level language code makes it easier to re-host the simulator (run it on a different platform, but simulating the same target architecture).

For run-time code generation, we produce a code generator for the *host* instruction set. The input to this code generator will be semantic trees representing the semantics of particular sequences of target instructions. We observe that here, as in the case of generating target code directly from Java bytecodes in the Jikes RVM "baseline" compiler, we would like to be able to generate code without actually constructing the semantic trees, so as to minimize the overhead of generating code at run time.

Timing Simulation: We will focus on timing simulation, built in the style of Schnarr [21]. In this style, the simulator maintains, in program execution order, a queue of the instructions currently in the modeled processor's pipelines, and their states. At each clock tick, the simulator scans the queue, from oldest to newest instruction, and tries to advance the state of each instruction. Some oldest instructions may retire and disappear from the queue, and some new instructions may enter the instruction fetch stage and be added to the queue. The time taken by the simulator is thus roughly proportional to the number of instructions executed times the average instruction's "lifetime" in the queue. Thus, performance is affected more by the depth of the pipelines than by instruction level parallelism (ILP).

The state of an instruction indicates where it is in the pipelines (at which functional unit, and where in that unit's processing of it). Because we advance instructions working from oldest to newest, we can do functional unit busy-ness bookkeeping very easily during the scan at each tick, reducing bookkeeping code and work.

To represent any instruction's need to wait for operands, we associate with the instruction one or more input and output events, represented as boolean flags indicating whether the events have occurred. When an instruction reaches a stage of execution (state) where it needs its inputs, the clock tick scan will check the input events, and will not advance the instruction's state if the necessary events have not yet occurred. Likewise, when an instruction enters the instruction queue, we clear its output events, and only when the instruction reaches the state in which the outputs are available do we set the events. We need not allocate and free these event structures, since we can associate them with the instruction's slot in the queue (i.e., keep the event flags in an array parallel to the instruction queue, which itself is a simple circular buffer). We also maintain a table that indicates, for each register, the latest output event that will write (or has written) that register. This is all actually quite efficient, as the following step-by-step description suggests:

1. When entering an instruction into slot k of the instruction queue, clear the output events associated with slot k.
2. For each input register r, associate with the instruction the *current event* of r.
3. For each output register r, record the appropriate output event of the current instruction as the current event of r. (This must happen *after* the previous step if the same register is an input *and* an output!)
4. When the instruction must wait for its operands, the state advancing code inhibits advance until the instruction's input event's are set to true.
5. When the instruction produces an output, it sets the corresponding output events to true.

To this we add some logic to handle the case when a register has not been updated in a long time (and the "current event" flag may be reused, which would confuse uses of the register), and for instructions executed speculatively but discarded—details we omit here.

Our point is that this kind of timing simulation is fairly general, and depending on the fetch logic and the interaction with other system components, can drive timing of a wide range of architectures. For example, we are convinced that not only can it be used for in-order and out-of-order pipelined superscalar RISC machines, including ones with "delay slots" and multiple levels of speculation through branches, it can also handle very long instruction word (VLIW) architectures, the predication and speculation of the Intel IA-64, etc. It is also suited to modeling the timing in multithreaded CPUs (the challenge there is the fetch logic).

Building timing simulators automatically: This leaves the important question: *How does one build such a simulator automatically?* The answer is that we describe the pipelines and the flows of the instructions through them. For this, others have proposed a variety of strategies, differing in their convenience, compactness, and checkability. One fairly recent approach that is helpful in our case is the *annotated pipeline graphs* of Christopher Milner [15], which he proposed for use in instruction schedulers. In his scheme one describes the pipelines and their connections, much as in architecture implementation descriptions from manufacturers. (The "annotations" express any additional scheduling constraints of the implementation.)

In Milner's scheme one further marks the pipeline elements with their *semantics*, i.e., what operation(s) they perform. (The inputs and outputs are determined by the pipeline graph structure.) We would express the semantics using the same semantic notation as we use for writing instruction semantics in ISA descriptions. Given the semantic annotations, we can then match each ISA instruction to the possible pipeline flows for that instruction. (We might have to rule out some possibilities as ones the control logic does not actually use.)

The advantage of this approach is, as Milner notes, that it decouples the descriptions of ISAs from the descriptions of architecture implementations, obviating the need to build a description for each (ISA, implementation) pair that one wants to work with. Coupling comes only through the use of a common semantic notation—the same notation that is in common between compiler IRs and the ISA description. When we generate a simulator we can check that each ISA instruction has a suitable pipeline implementation, etc.

Speeding Timing Simulation Using State Transition Caches: Recent work by Schnarr [21, 23, 22] demonstrated that, with a suitable encoding of machine states, one can cache (memoize) machine state transitions and substantially speed up timing simulation.

Schnarr's FastSim [22] and Facile [21, 23] systems build up a cache of state to state transitions. He found that, given enough memory, this cache significantly speeds up timing simulation. We can build such a caching system into our design.

Schnarr's Facile system analyzes the target executable in advance, and generates simulation code for every basic block. It also caches state transition information for every basic block executed (though it can discard and reconstruct that information if the state transition cache grows too large). We intend to produce run-time code only for frequently executed blocks, and to apply adaptive optimization to obtain better code for the most frequently executed blocks. We will have at our disposal a system that can apply a range of sophisticated optimizations, if the cost is warranted. We also intend to separate functional and timing simulation, which we believe will lead to better speedup, and in any case make it easier to recover when we mispredict execution paths. (Schnarr's system must make special provision to avoid re-executing that part of functional (and timing) simulation that was done up to the point that misprediction was detected, which considerably complicates the design and slows the system down.)

5. A Prototype to Begin Realizing the Vision

Having laid out our vision and some of the general research problems involved, we now describe more concretely the prototype we aim to build, illustrated in Figure 1. The figure shows the simulator and compiler frameworks at the sides, emphasizing that generated components "plug in" to the frameworks. The center column shows the ISA, IR, and pipeline descriptions (and the instrumentation aspect). The lines and arrows show which descriptions are used to build which generated components. To avoid visual clutter, the figure does not show the individual generator programs and tools (except for AspectJ).

We intend to do most of the work in the context of the Jikes RVM Java virtual machine. We have considerable experience using and modifying the Jikes RVM, which was originally called Jalapeño and was developed by a team at IBM Research. It is now available as an open source system, with many academic research groups using it for work in compilation.

Jikes RVM is good for automatically retargeting both compiler back-ends *and* simulators. There are, as with any system, some system-specific and practical issues we face in using the Jikes RVM for this work. They include:

- While the optimizing compiler is designed to be retargeted using BURS rules, which we intend to generate from ISA semantic descriptions, the baseline compiler is not designed in a retargetable way. We need to figure out how to generate an efficient baseline compiler from semantic descriptions of the Java bytecodes (the "IR" in this case) and the target ISA.
- For the optimizing compiler we will need to make register allocation automatically retargeted. This may involve developing automatically generated calling convention descriptions—which can be used in both the baseline and optimizing compilers.
- Java is not viewed as being as efficient a vehicle for simulation as C and other unsafe languages. The Jikes RVM can *selectively* suppress checks that we know are unnecessary. We believe that this, combined with the quality of the Jikes RVM optimizers, will enable us to produce a system with competitive performance ... and with substantial software engineering benefits over C systems. In brief, we will be able to build, and *debug*, a system faster. We already have this experience in using Java to write a whole series of garbage collectors for the Jikes RVM.
- We will need to develop a simulation framework somewhat from scratch. We can base it on existing systems, such as SimpleScalar, but the automatic retargeting that is the essence of our approach means that many pieces will be new. But we can certainly exploit previous tools and languages for *generating* these components, retargeting them to produce Java code to interface with our frameworks.

5.1. Prototype Description Languages

We previously mentioned the MLRISC system, which allows one to describe the "syntax" of instructions (bit fields, opcode values, etc.), and to relate the bit syntax to assembly code syntax (to build assemblers, disassemblers, and target-specific debugger modules). More significantly, one can associate semantics with the syntax, in the form of trees. MLRISC further provides for register allocation information. MLRISC collaborators have already generated machine descriptions for a range of modern CPUs, including the Alpha (32 and 64 bit), HPPA, IA-32, PowerPC (32 and 64 bit), and SPARC.

The MLRISC project aims to produce quality back-ends automatically from machine descriptions. They do so by specializing a "generic" optimizing back-end to the target instruction set. They assume a front-end that produces MLRISC trees. Their code generator (instruction selection) strategy is simpler than a BURS rule system (but they say it is more efficient). Given our compiler and simulation framework, we cannot use MLRISC components directly, but one strategy would be to modify their *mdgen* program, which generates MLRISC target-specific compiler components from a machine description, to generate components for our system.

One admitted weakness of MLRISC is in the area of calling conventions and register usage, where it may help to introduce additional descriptions along the lines of Bailey and Davidson, as previously mentioned.

5.2. Prototype Compiler Framework

The parts of the Jikes RVM that need retargeting, include the object and class layout portion of the class loader, the generation of BURS rules, the instruction scheduler, the register allocator, the assembler (constructs machine code words/bytes from MIR form), the disassembler (used for producing listings and debugging output), and the baseline compiler and its assembler. Most of these pieces of a compiler have been automatically retargeted before, so we build on the others work quite directly. Generating BURS rules is a little different and perhaps the most challenging part, as previously discussed, but we can start with Cattell's work

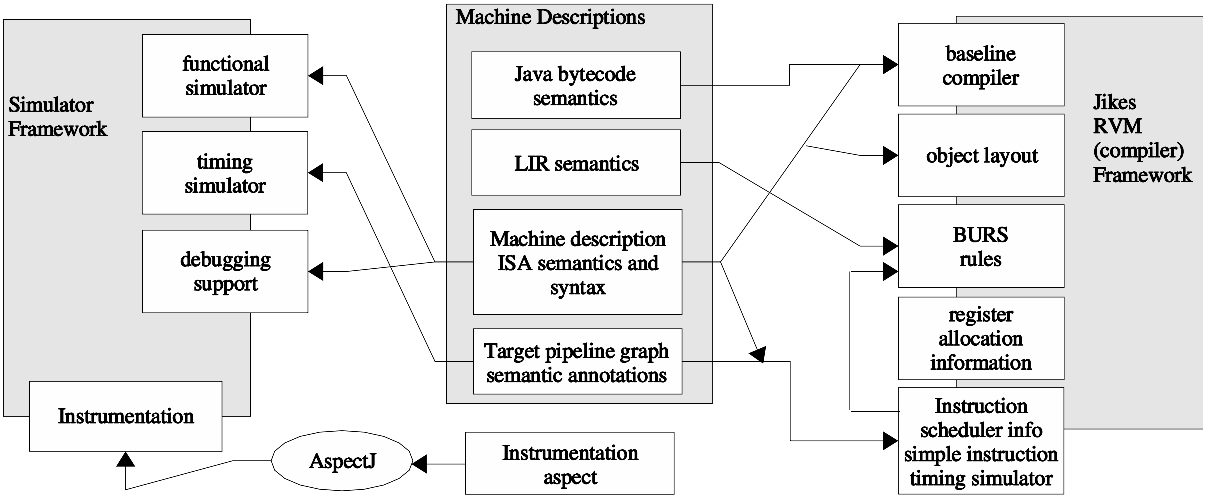

Figure 1. Architecture of the Proposed Prototype System

[5]. We also need timing estimates for rule costs, which we can obtain from a stripped down timing simulator.

Challenging but manageable: We feel that the project is, on the one hand, a significant enough advance in the state of the art to be worthwhile, and on the other hand, enough of it consists of synthesis and extension of previous work to make it manageable. Of particular note is the hundreds of thousands of lines of code in the Jikes RVM that we are exploiting in not building a JVM and JIT compiler(s) from scratch. The practical side of the work consists mostly in making a fairly retargetable system more thoroughly retargetable. We also note that in making the Jikes RVM automatically retargetable, our whole platform becomes widely portable as a "side-effect" of our work, substantially increasing its utility to others.

5.3. Prototype Simulator Framework

As we mentioned above, while the compiler framework mostly exists, the simulator framework will be mostly new, but we will gain significant benefits by building it in Java and in the Jikes RVM in particular, which we will not reiterate here. However, we do have a few additional observations:

- Dynamically generating simulation code is very similar to performing dynamic binary translation, so a fairly easily achieved additional result of our work would be an automatically retargeted dynamic binary translation system.
- We may need to develop some interesting extensions to the Jikes RVM and its compilers in order to realize all the simulation speedup ideas we envision. For example, at present, every piece of dynamically generated code is a Java method. How will we generate thousands of code snippets, which may not ever exist in Java bytecode form, and integrate them into the Jikes RVM system (including garbage collection, exception handing, thread switching, etc.)? Likewise, we may want to add an efficient co-routining mechanism, or use threaded code, both of which require system support, and possibly compiler extensions. The Jikes RVM will be a good vehicle for such exploration, though, given its completeness, relatively modular structure, and the community of capable people working with it.
- In addition to MLRISC for ISA syntax and semantics, we will develop and implement a language for expressing pipelines and their semantics, based on Milner's proposal [15] as previously discussed.

6. Conclusion

We have described the need for cycle-level timing simulators to evaluate new architectural features and ideas in the context of modern programming languages, and the need for corresponding optimizing compiler back-ends. Further, these should be generated automatically from precise and concise descriptions, both to speed architectural exploration and to prevent errors and bugs. Enough prior work has been done on the component problems that the proposed work, while challenging, can be accomplished. We expect the resulting system to be widely portable and readily used by others in research, and it will also have many uses in teaching about modern compilers, architectures, and simulators.

References

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, Oct. 1989.

[2] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño—a compiler-supported Java virtual machine for servers. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software*, Atlanta, GA, May 1999. ACM.

[3] M. W. Bailey and J. W. Davidson. Construction of systems software using specifications of procedure calling conventions. Submitted for publication.

[4] M. W. Bailey and J. W. Davidson. A formal model and specification language for procedure calling conventions. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–310, San Francisco, CA, 1995. ACM.

[5] R. G. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190, Apr. 1980.

[6] D. C.Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997. Extended version available as Univ. of Wisc. Comp. Sci. Tech. Rep. 1342, June, 1997.

[7] T. Elrad, R. E. Filman, and A. Bader. Aspect oriented programming. *Commmunications of the ACM*, 44(10):29–38, Oct. 2001.

[8] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sept. 1992.

[9] L. George and A. Leung. MLRISC: A framework for retargetable and optimizing compiler backends. Technical report, Bell Laboratories and New York University, 2000.

[10] X. Huang, J. E. B. Moss, K. S. McKinley, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java virtual machines. In preparation., 2002.

[11] G. Kiczales et al. An overview of AspectJ. In *Proceedings of the Fifteenth European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2001.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[13] F. Larsson. Generating efficient simulators from a specification language. Master's thesis, Computing Science Department, Uppsala University, Uppsala, Sweden, 1997. Published as thesis number 1997-01-29.

[14] H. Massalin. Superoptimizer—a look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, CA, 1987.

[15] C. W. Milner. Pipeline descriptions for retargetable compilers: A decoupled approach. Technical Report CS-99-11, University of Virginia, June 1998.

[16] M. Moudgill, J.-D. Wellman, and J. H. Moreno. Environment for PowerPC microarchitecture exploration. *IEEE MICRO*, 19(3):15–25, May/June 1999.

[17] S. R. of Machine Instructions. Norman ramsey and mary f. fernandez. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[18] N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98)*, pages 172–188, June 1998. Available as Springer Verlag LNCS 1474.

[19] N. Ramsey and M. F. Fernandez. Automatic checking of instruction specifications. In *1997 International Conference on Software Engineering*, pages 326–336, May 1997.

[20] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, Fall 1995.

[21] E. Schnarr, M. D. Hill, and J. R. Larus. Facile: A language and compiler for high-performance processor simulators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2001.

[22] E. Schnarr and J. R. Larus. Fast out-of-order simulation using memoization. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. ACM, Oct. 1998.

[23] E. C. Schnarr. *Applying Programming Language Implementation Techniques to Processor Simulation*. Ph.d. dissertation., Computer Sciences, University of Wisconsin–Madison, 2000.