Just-in-Time Compilation for FPGA Processor Cores

Andrew Becker, Scott Sirowy, Frank Vahid
Department of Computer Science and Engineering
University of California, Riverside
{abecker | ssirowy | vahid}@cs.ucr.edu

Abstract

Portability benefits have encouraged the trend of distributing applications using processor-independent instructions, a.k.a. bytecode, and executing that bytecode on an emulator running on a target processor. Transparent just-in-time (JIT) compilation of bytecode to native instructions is often used to increase application execution speed without sacrificing portability. Recent work has proposed distributing FPGA circuit applications in a SystemC bytecode to be emulated on a processor with portions possibly dynamically migrated to custom bytecode accelerator circuits or to native circuits on the FPGA. We introduce a novel JIT compiler for bytecode executing on a soft-core FPGA processor. During an iterative process of JIT compiler and emulator architecture codesign, we added JIT-aware resources on a soft-core processor's surrounding FPGA fabric, including a JIT memory, a signal queue, and an emulation memory controller—all unique to JIT compilation for FPGA processors versus traditional processors. Experiments show that regular JIT compilation achieved 3.0x average speedup over emulation, while our JIT-aware FPGA resources yielded an *additional* 5.2x average speedup, for a total of 15.7x average speedup, at a cost of 21% of a MicroBlaze processor core's slice usage.

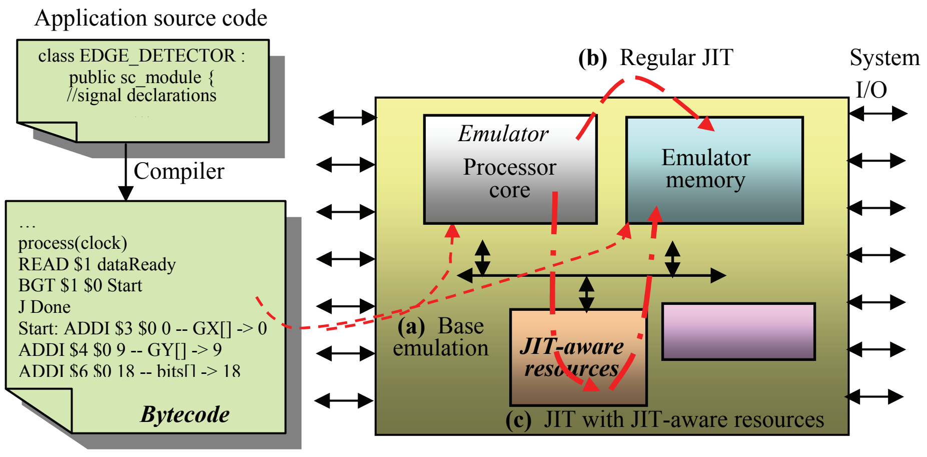
1. INTRODUCTION

Application circuit designers have traditionally used simulation for prototyping and verification. However, simulation does not allow in-system execution, which uses actual external inputs and outputs. For that, designers have used field-programmable gate arrays (FPGAs). While FPGAs assist with prototyping, long synthesis times of hours inhibit rapid iterations in the design process.

To help alleviate this problem, Sirowy [22] developed SystemC bytecode. A custom SystemC compiler generates bytecode in just seconds, consisting of processes each with bytecode-described behavior, and inter-process signals. A bytecode emulator on an FPGA processor core, the emulator consisting of an event-driven simulation kernel, executes the bytecode. By using SystemC rather than languages like C or C++, designers can express their designs in a manner similar to their final design. Only the synthesizable subset of SystemC is supported by the compiler, so the same code used for emulation may be synthesized directly with little modification. Choosing bytecode over native code has the advantages both of portability and shifting optimization of execution speeds to the emulation platform's designer. SystemC bytecode enables rapid prototyping with minimal extra work required from designers, and is useful for teaching SystemC without requiring synthesis tools.

However, SystemC bytecode emulation has a large performance overhead versus synthesized circuits. Sirowy added accelerators with dynamic process mapping algorithms [21] for emulator speedup. We found that just-in-time (JIT) compilation of the bytecode to the emulator's processor could yield several times speedup, as illustrated in Figure 1. We thus developed a JIT compiler for the emulator's processor core on the FPGA. Furthermore, the processor's surrounding FPGA fabric provides

Figure 1: JIT compilation for an FPGA processor core: (a) initial application executes on an emulator executing on an FPGA processor core; such emulation may be slow, (b) regular JIT compilation yields good speedups of 3x, (c) FPGA configurability enables introduction of JIT-aware resources, which improves speedup by another 5.2x, for a total of 15.7x speedup.



unique opportunities for the JIT compiler. The contribution of this work is the creation of the *first JIT compiler for an FPGA processor core*, including introduction of *JIT-aware resources* to yield several times additional speedup beyond traditional JIT compilation. These resources were created via an iterative JIT compiler/architecture codesign approach that is possible on FPGAs.

2. RELATED WORK

Much previous work focuses on dynamic binary translation and JIT compilation to improve performance of software interpretation on a processor. JIT compilation often improves execution to near native speeds [12]. The Transmeta Crusoe processor [7] dynamically translated x86 code into native VLIW instructions for improved performance and reduced power. Other architectures, like accumulation-based computer architectures [13], have also benefited from JIT compilation techniques. Gligor [10] used dynamic binary translation to improve the speed and flexibility of MPSoC simulations. Brandner [6] develops a lightweight compiler for Java bytecode intended for small embedded processors.

Some previous work investigates capturing applications and circuits to increase portability. Andrews [2][3] focuses on creating operating system and middleware abstractions that extend across the hardware/software boundary, enabling a designer to create applications for hybrid platforms with one executable. Levine [14] describes hybrid architectures with a single, transformable executable. They argue that an executable described for a queue machine (converse of a stack machine) makes runtime optimizations to a specialized FPGA fabric feasible. Moore [17] describes writing applications that dynamically bind at runtime to reconfigurable hardware for portability. Similar to Andrews [3], the authors develop hardware/software abstractions by writing middleware layers that allow application software to utilize reconfigurable DSP cores. Vuletic [25] proposes a system-level virtualization layer and a hardware-agnostic programming paradigm to hide platform details from the application designer and lead to more portable circuit applications.

Much work improves virtualized software execution using supporting hardware. Intel and others added instructions to the x86 architecture to improve virtualization performance [18]. Adams [1] presents a survey of techniques for improving x86 virtualization execution, discussing both software and hardware optimizations. Rosenblum [20] discusses the advantage of hardware-level virtual machines, and the need to make them as fast, efficient, and transparent as possible. Enzler [9] uses reconfigurable arrays to virtualize hardware. Bauer [4] uses reconfigurable arrays to improve the execution time of event-driven simulation. Some work has focused on accelerating Java bytecode through the design of custom bytecode accelerators [11][18][6].

Vahid [24] introduced warp processing, wherein processor binary (native or bytecode) kernels are JIT synthesized to FPGA coprocessor circuits for speedup, using an FPGA fabric designed for fast synthesis. Bergeron [5] JIT synthesized binary kernels to custom instructions implemented on a reconfigurable region of a Xilinx FPGA fabric. These JIT techniques compile FPGA processor binaries to circuits rather than to native processor code as in this paper.

Sirowy introduced SystemC bytecode and emulation [22] along with accelerators [21] (which could be further supplemented with warp processing). Our work focuses on JIT compiling bytecode to the emulator's processor for speedup of code running on the emulator (before or after acceleration or warp processing). Though the paper focuses on SystemC bytecode, some of the JIT-aware resource techniques could be applied to any emulation approach that uses an FPGA processor, such as Java bytecode emulation.

3. SYSTEMC BYTECODE

Our framework applies JIT compilation to speed up SystemC bytecode emulation. SystemC [23] uses object-oriented features of C++ to enable descriptions that include features common in previous hardware description languages (HDLs), such as creation of components, instantiation and connection of components to form a circuit, and precisely-timed communication and execution among concurrently-executing components, all using existing C++ syntax. Regular C++ code can be included in descriptions, and SystemC also provides a thread library, thus supporting description of both the "software" (sequential instructions coupled with parallel threads) and "hardware" (circuit) parts of an entire system in a single description language.

A SystemC description is typically executed on a PC for simulation purposes, using simulated inputs/outputs (I/O). Later, the description may be synthesized onto an ASIC, FPGA, or PC board, using real I/O for "in-system" execution. The synthesis process compiles parts of the description to microprocessors and synthesizes other parts to circuits. Synthesis can be time-consuming, requiring hours or days, depending on the description's complexity, available tools, and designer expertise. Sirowy [22] developed an approach wherein the original description can be compiled in seconds to a newly-developed SystemC bytecode format, and then executed on an emulator that interacts with real-world I/O, thus quickly obtaining in-system execution, at the expense of greatly reduced execution speed. Such an in-system emulation approach can support early testing, and may even be a sufficient implementation for some applications not requiring high speed. To speed up emulation on an FPGA processor core, Sirowy introduced bytecode accelerators [21]. These are customized processors in the FPGA fabric designed to natively execute bytecode faster than the emulator on the processor core would be able to. Speedups ranged from 10x-100x, but required a large amount of FPGA resources.

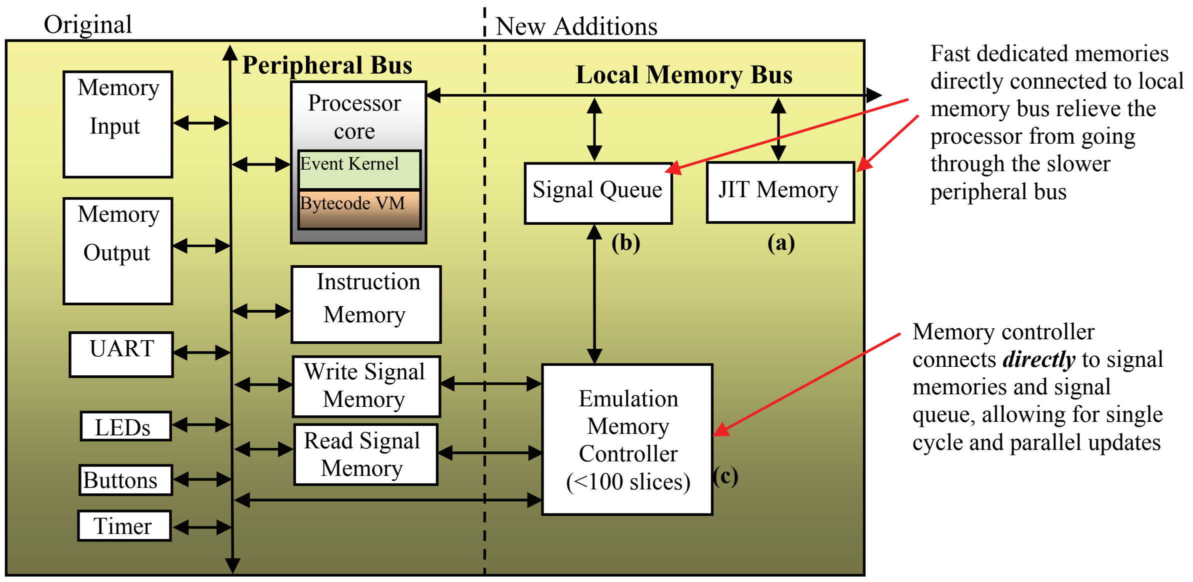
4. BASE EMULATOR

For the given bytecode, we utilized a previously-developed base emulator [22]. The emulator is written in 3,000 lines of C and consists of a lean event-driven kernel, a virtual machine to execute the bytecode instructions, and hardware resources and hooks for certain peripherals on the FPGA. We implemented the emulator on a MicroBlaze soft-core processor in a Virtex5 VLX110T, with the emulator executing from a large SRAM. Two small scratchpad memories are used on the FPGA for fast access to particular data items used by the emulator. The FPGA architecture for the emulator on the MicroBlaze appears on the left of Figure 2.

We ran several benchmark applications on the emulator. We chose applications with varied amounts of complexity: An edge detection benchmark (with heavy peripheral I/O), an A5/1 cipher benchmark (a basic implementation of an encryption algorithm used in GSM phones), a matrix multiplication application, a digital timer, and a sequencer implemented as a simple state machine.

To compare the speed of JIT compiled code with an "upper bound," we manually wrote each benchmark as a C program (not SystemC) implementing the same behavior. The C programs are less intuitive than the SystemC descriptions and the parallelism in the application is less exposed, but the C descriptions provide an

Figure 2: JIT-aware resources: (a) JIT memory, (b) signal queue, (c) emulation memory controller.


Original
New Additions
Peripheral Bus
Local Memory Bus
Memory Input
Memory Output
UART
LEDs
Buttons
Timer
Processor core
Event Kernel
Bytecode VM
Instruction Memory
Write Signal Memory
Read Signal Memory
Signal Queue
JIT Memory
(b)
(a)
Emulation Memory Controller (<100 slices)
(c)
Fast dedicated memories directly connected to local memory bus relieve the processor from going through the slower peripheral bus
Memory controller connects *directly* to signal memories and signal queue, allowing for single cycle and parallel updates

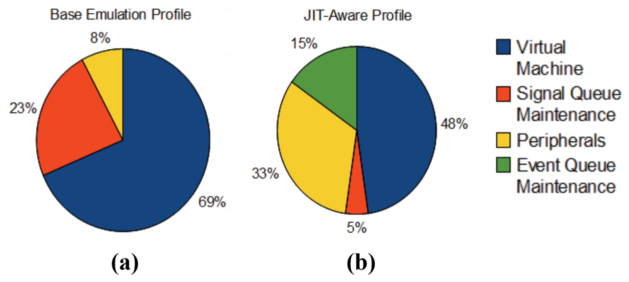
upper bound to how fast the bytecode could possibly execute on a MicroBlaze—essentially, the C code strips away all SystemC overhead and describes just the application code. We compiled the C descriptions directly to MicroBlaze machine code using the Xilinx tools and the highest levels of optimization (O3). We refer to this implementation as '*Native C'*.

We built multiple versions of the platform, both with the dedicated JIT architecture changes and without. The emulation architectures with dedicated JIT architecture changes were described using approximately 10,000 lines of VHDL. The emulation architectures were built using Xilinx ISE 11, and the software was compiled using Xilinx EDK 11.

5. JIT COMPILATION AND JIT-AWARE RESOURCES

We profiled the emulator's execution for the benchmarks. Figure 3(a) shows that virtual machine execution amounts to 69% of the emulator's execution time; the other contributors relate to architectural features. The virtual machine's dominance is due to the large overhead of software interpretation of bytecode instructions. JIT compilation from bytecode instructions to MicroBlaze instructions should thus greatly decrease that time,

Figure 3: Emulator execution profile: (a) initial, (b) after JIT-aware modifications.



because almost all of the RISC-like bytecode's instructions can be translated to just two MicroBlaze instructions on average.

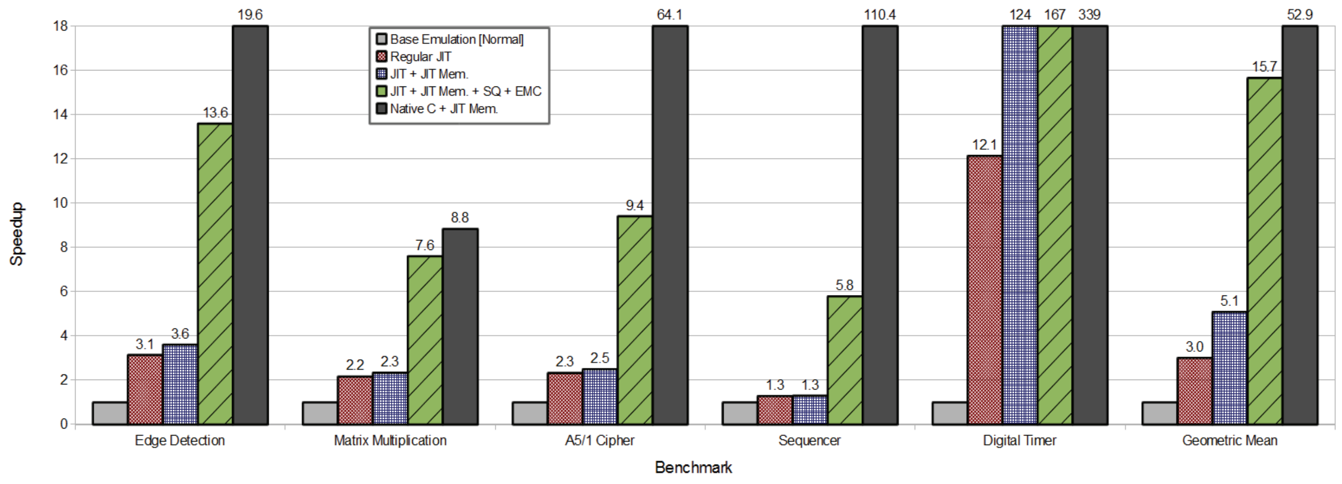
5.1 Regular JIT compilation

JIT compilation from the bytecode to the target platform is straightforward, consisting of three analysis phases and three translation passes. The first analysis phase determines how many instructions each source bytecode instruction will require in the target architecture, i.e., the size of the MicroBlaze instruction sequence required to implement each bytecode instruction. The second and third analysis phases determine which bytecode registers depend upon values from previous executions of the process (a feature required by the SystemC emulator) and which MicroBlaze architecture conventions might be violated by naïvely translated code – any registers that must be saved across function calls should not be overwritten by process execution, the stack pointer must remain unchanged, etc.

The first translation pass copies bytecode instructions to appropriate locations in the JIT memory, calculated from the information gleaned in all three analysis phases. The first analysis phase's information gives the size of each MicroBlaze instruction sequence, while the second and third phase information is used to determine how much space to reserve at the beginning and end of the destination memory space. The second pass translates each bytecode instruction to its corresponding MicroBlaze instruction sequence. The translation requires two distinct tasks: (1) relative and absolute branches must be recalculated, and (2) each bytecode instruction must be expanded into its equivalent sequence of MicroBlaze instructions. The third pass adds a function prologue and epilogue to ensure compliance with the emulation engine and architecture register conventions, which requires the information from the latter two analysis phases.

We implemented the JIT compilation routines in approximately 1,700 lines of C. Figure 4 shows the speedup obtained via a simple JIT compilation compared to base emulation, ranging from 1.3x to 12.1x, with a geometric mean of 3.0x. (Geometric mean is less influenced by outliers than arithmetic mean).

Figure 4: Speedups of various JIT implementations normalized to execution on the base emulator. “Regular JIT” places JIT-compiled native instructions into the same memory as the emulator (SRAM). “JIT + JIT Mem” adds a dedicated small fast memory (FPGA block RAM) to store native instructions. “JIT + JIT Mem + SQ + EMC” further adds dedicated signal queues and an emulator memory controller. “Native C + JIT Mem” is an upper-bound for JIT compilation, achieved by manually writing C code and compiling directly to native instructions, stored in a dedicated small fast memory.



5.2 Adding a JIT memory resource

Regular JIT compilation yields good speedups, but greater speedups are possible. One potential area of improvement is the emulator memory architecture. The emulator requires a large instruction memory, heap, and stack. However, FPGAs typically do not have large on-chip memories, but rather numerous smaller memories. Implementing the emulator’s memory on numerous small on-chip memories results in poor performance, may exhaust available memory resources, and prevents other circuits mapped to the FPGA fabric from using memory resources. Thus, the emulator’s memory usually needs to reside in a larger, slower memory (generally DRAM or SRAM). Naïvely placing the relatively small native code resulting from JIT compilation back into this same memory yields performance improvement, but this improvement is hampered by memory latency.

We observed that since the native code resulting from JIT compilation is much smaller than the emulator memory, the emulator would benefit from using a small but fast memory dedicated for storing the JIT compiled native code, as shown in Figure 2(a). We implemented the dedicated JIT memory and signal queue in FPGA block RAM, but it could also be implemented in distributed FPGA RAM. The dedicated JIT and signal memories directly connect to the MicroBlaze via a local memory bus. The JIT memory can hold small amounts of natively translated bytecode (we used 64KB of BRAM to implement this memory, but designers could use any size which suits them), but can execute orders of magnitude faster than the virtual machine interpreting bytecode. The JIT compiled code in this dedicated memory is also several times faster than translated code executed from the original slower memory.

For our implementation, we assume the emulator can JIT compile the *entire* process to the dedicated JIT memory, but each process may be JIT compiled or not independently of other processes. In addition, because the emulator architecture manages JIT compilation of each process separately, processes may be dynamically swapped in and out of the dedicated JIT memory, though this is not currently implemented. The emulator can fall back on JIT compiling the bytecode to the larger, slower memory resources and still see performance improvement, if necessary.

For each application, the emulator JIT compiles the processes in the SystemC circuit *prior* to emulation execution. The rationale for this is JIT compilation during emulation would introduce timing uncertainties which are generally unacceptable for embedded applications. For instance, in the Sobel example, a jittery video display may be less desirable than a few extra milliseconds to start up. However, if on-the-fly JIT compilation is desired, modifying the emulator to translate the processes during emulation is simple. A flag could be set in the event record blocking emulation of the process undergoing JIT compilation, bytecode register values that must be saved could be copied to the appropriate locations in the JIT memory, and then the flag blocking execution would be unset. In either case, the time required to JIT compile is a one-time expenditure and runs in milliseconds, even for large multi-process applications. If on-the-fly JIT compilation is required but timing uncertainties are still an issue, each analysis and translation phase could be executed separately when resources are available, and each phase itself could be split to process only one bytecode instruction at a time, thus reducing jitter.

Figure 4 shows that adding the dedicated JIT memory sometimes achieves big additional speedups, while at other times yields no speedup, depending upon whether instruction memory latency is a large bottleneck for a benchmark. For computationally intensive benchmarks, where JIT instruction memory latency is a much bigger factor than virtual-machine/architecture overhead, a dedicated JIT memory makes a big impact. For example, in the digital timer benchmark, JIT compilation to dedicated memories resulted in a 10x speedup compared to regular JIT compilation. For other benchmarks, like the sequencer benchmark, the speedups were less impressive, yielding essentially no speedup over regular JIT compilation. The geometric mean of the speedup was 5.2x.

5.3 Adding a dedicated signal queue and an emulation memory controller resource

Having addressed the emulator memory architecture, Figure 3(a) shows that the next area for potential improvement involves the management of signal queues. In SystemC, processes communicate through signals, which are akin to global variables with specific time-stamps associated with values. An executing process may schedule a signal for update; after all processes execute, the emulator then examines each signal's queue and updates the signal value appropriately. A SystemC description may have hundreds or thousands of signals, each with its own queue, and thus writing and reading such queues contributes to increased execution time.

Thus, an improvement is to create a component on the FPGA fabric specifically dedicated storing signal queues, as shown in Figure 2(b). The original emulator implementation managed the signal queue dynamically in software, making numerous low-level memory allocation calls. We observed that the signal queue is bound in size by the size of the read and write signal memories, and thus decided that a statically-instantiated fast memory would mitigate the effect of dynamic signal queue management and reduce memory latency on all signal queue accesses. A 16KB signal queue sufficed for our implementation, but designers may wish to use a different size depending upon their needs. We also observed that instead of *enqueueing* and *dequeuing* signals to and from the signal queue, the emulation engine need only store a signal identifier, reducing the overhead on the bus to which the signal queue is attached to the processor.

Furthermore, we observed that we could offload the tasks of updating the memories and the maintenance of the signal queue to a dedicated emulation memory controller component implemented on the FPGA. Upon completion of execution of active processes, the emulator commands the emulation memory controller to update the signal memories and populate the signal queue. The emulation memory controller iterates over the write signal memory, finds any signals that have been updated, updates the read memory signal value, and adds the updated signal to the signal queue. These actions can be pipelined, meaning that the emulation memory controller can check, update, and enqueue every signal in the system in one pass. For a SystemC application with 100 signals, the emulation memory controller can finish updating all signals in 100 clock cycles. In contrast, a software approach requires many hundreds or thousands of cycles to carry out the same behavior.

Figure 4 shows the additional speedups gained by addition of the dedicated signal queue memories and the emulation memory controller. Benchmarks with more accesses to signals require more time managing signal queues, so those benchmarks achieve a larger benefit than those with fewer accesses. For example, the sequencer benchmark achieved a 5.8x speedup, versus just 1.3x for the regular JIT even with a dedicated memory. The geometric means of speedup was 15.7x.

We note that, because a hard core processor is typically faster than a soft core processor, the additional speedups via JIT-aware FPGA resources could be less for a hard core than for a soft core due to those resources running slower relative to the core, but this possibility remains to be investigated.

Figure 4 also shows "upper bound" data for natively-written C compiled directly to native processor instructions. For some benchmarks, JIT compilation approaches the bound, but others show 5x to nearly 20x additional speedup unachieved. Future work may reveal whether those speedups could be attained by more aggressive JIT compilation that goes beyond instruction-level translation, and instead uses more aggressive decompilation/recompilation methods.

6. CONCLUSIONS

While regular JIT compilation of SystemC bytecode for a Xilinx MicroBlaze soft-core processor achieved average speedups of 3.0x and ranged from 1.3x to 12.1x speedups, adding JIT-aware FPGA resources achieved average speedups of 15.7x and ranged from 5.8x to 167.0x. Adding JIT-aware FPGA resources thus improved JIT compilation by an average of 5.2x, and by as much as 13.9x. The logic cost of such resources, including bus logic, is 75 FPGA slices, which corresponds to 21% of the size of the present MicroBlaze processor core. Block RAM cost for our implementation was 80 KBytes, though this was more than enough for even the largest of our benchmarks, and designers will wish to vary block RAM allocation to suit their platform.

Bytecode accelerators, which are custom soft-core processors whose native instruction set is SystemC bytecode, were previously shown to provide 10x-100x speedups [21], but require more resources than JIT compilation, starting with several hundred slices for one accelerator, and requiring multiple accelerators for speedups to scale to multiple SystemC processes. Thus, JIT compilation should be incorporated first, and then accelerators added if they yield additional benefit.

JIT compilation with JIT-aware resources offers an attractive performance improvement for relatively small amounts of FPGA resources. Future work may include integration of JIT compilation with bytecode accelerators in a single platform for more design flexibility and hence greater overall speedups, and integration with JIT synthesis to custom coprocessor circuits.

7. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation (CNS1016792) and by the Semiconductor Research Corporation (GRC 2143.001).

REFERENCES

[1] ADAMS, K. AND AGESEN, O. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems* (San Jose, California, USA, October 21 - 25, 2006).

[2] ANDERSON, E., AGRON, J., PECK, W., STEVENS, J., BAIJOT, F., KOMP, E., SASS, R., AND ANDREWS, D. 2006. Enabling a Uniform Programming Model Across the Software/Hardware Boundary. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (April 24 - 26, 2006). FCCM

[3] ANDREWS, D., SASS, R., ANDERSON, E., AGRON, J., PECK, W., STEVENS, J., BAIJOT, F., AND KOMP, E. 2008. Achieving programming model abstractions for reconfigurable computing. *IEEE Trans. Very Large Scale Integr. Syst.* 16, 1 (Jan. 2008), 34-44.

[4] BAUER, J., BERSHTEYN, M., KAPLAN, I., AND VYEDIN, P. 1998. A reconfigurable logic machine for fast event-driven simulation. In *Proceedings of the 35th Annual Design Automation Conference* (San Francisco, California, United States, June 15 - 19, 1998). DAC '98.

[5] BERGERON, E., M. FEELEY, AND J.P. DAVID. Hardware JIT compilation for off-the-shelf dynamically reconfigurable

FPGAs. Int Conf on Compiler Construction (CC'08/ETAPS'08), Berlin, 2008, pp. 178-192.

[6] BRANDNER, F., T. THORN, AND M. SCHOEBERL. Embedded JIT compilation with CACAO on YARI. In *Processdings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing* (Tokyo, Japan, March 17 – 20, 2009). ISORC 2009.

[7] DEHNERT, J. C., GRANT, B. K., BANNING, J. P., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSON, J. 2003. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*

[8] ECMA. Rue du Rhone 114 CH-1204 Geneva. *Standard ECMA-335 Common Language Infrastructure (CLI)*, 3rd edition, June 2005.

[9] ENZLER, R. PLESSL, C. AND PLATZNER, M. Virtualizing Hardware with Multi-Context Reconfigurable Arrays. Lecture Notes in Computer Science. Springer Publishing. September 2003.

[10] GLIGOR, M., FOURNEL, N., AND PÉTROT, F. 2009. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In *Proceedings of the 7th IEEE/ACM international Conference on Hardware/Software Codesign and System Synthesis* (Grenoble, France, October 11 - 16, 2009). CODES+ISSS '09

[11] GRUIAN, F. AND WESTMIJZE, M. 2007. BlueJEP: a flexible and high-performance Java embedded processor. In *Proceedings of the 5th international Workshop on Java Technologies For Real-Time and Embedded Systems* (Vienna, Austria, September 26 - 28, 2007). JTRES '07, vol. 231. ACM, New York, NY, 222-229

[12] KAZI, I. H., CHEN, H. H., STANLEY, B., AND LILJA, D. J. 2000. Techniques for obtaining high performance in Java programs. *ACM Comput. Surv.* 32, 3 (Sep. 2000), 213-240.

[13] KIM, H. AND SMITH, J. E. 2003. Dynamic binary translation for accumulator-oriented architectures. In *Proceedings of the international Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (San Francisco, California, March 23 - 26, 2003).

[14] LEVINE, B. A. AND SCHMIT, H. H. 2003. Efficient Application Representation for HASTE: Hybrid Architectures with a Single, Transformable Executable. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (April 09 - 11, 2003). FCCM. IEEE Computer Society, Washington, DC, 101

[15] LEVIS, P. AND CULLER, D. 2002. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.* 36, 5 (Dec. 2002), 85-95

[16] MICROSOFT CORPORATION. *Common Language Runtime Overview.* http://msdn.microsoft.com/en-us/library/ddk909ch.aspx

[17] MOORE, N., CONTI, A., LEESER, M., AND KING, L. S. 2007. Writing Portable Applications that Dynamically Bind at Run Time to Reconfigurable Hardware. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (April 23 - 25, 2007). FCCM. IEEE Computer Society, Washington, DC, 229-238

[18] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., UHLIG, R. *Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization*. http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art01.pdf

[19] PARNIS, J. AND LEE, G. 2004. Exploiting FPGA concurrency to enhance JVM performance. In *Proceedings of the 27th Australasian Conference on Computer Science - Volume 26* (Dunedin, New Zealand). Estivill-Castro, Ed. ACSC, vol. 56. Australian Computer Society, Darlinghurst, Australia, 223-232

[20] ROSENBLUM, M. 2004. The Reincarnation of Virtual Machines. *Queue* 2, 5 (Jul. 2004), 34-40.

[21] SIROWY, S., C. HUANG, AND F. VAHID. Online SystemC Emulation Acceleration. IEEE/ACM Design Automation Conference, June 2010

[22] SIROWY, S. AND VAHID, F. Portable SystemC-on-a-Chip. International Conference on Hardware-Software Codesign and System Synthesis CODES+ISSS 2010.

[23] SYSTEMC. http://www.systemc.org

[24] VAHID, F., G. STITT, AND R. LYSECKY. Warp Processing: Dynamic Translation of Binaries to FPGA Circuits. IEEE Computer, Vol. 41, No. 7, July 2008, pp. 40-46.

[25] VULETIC, M., POZZI, L., AND IENNE, P. 2005. Seamless Hardware-Software Integration in Reconfigurable Computing Systems. *IEEE Des. Test* 22, 2 (Mar. 2005), 102-113