A Runtime Profiler: Toward Virtualization of Polymorphic Computing Platforms

Hamid Mushtaq, Mojtaba Sabeghi, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{*H.Mushtaq, M.Sabeghi, K.L.M.Bertels*}*@tudelft.nl*

***Abstract*—Runtime multitasking support on Reconfigurable Computers requires complicated resource management techniques in which the FPGA area has to be shared between multiple concurrent tasks dynamically. Such a resource allocation mechanism needs to know the current configuration and load of the system in order to decide about the allocation of the resources. In such systems, A runtime profiler is an important tool which can give vital information about the running applications on the system. In this paper, we present the design and implementation of a runtime profiler which is responsible to produce statistics about the code running on the system. We have performed a set of experiments in order to show the overhead of our proposed profiler. The evaluation results show that the overhead imposed by the profiler is less than 1.5% of the total execution time and the information generated by the profiler is almost as accurate as a design time profiler such as *gprof*.**

I. Introduction

Limited commercial success of Reconfigurable systems has been due to difficulty in programming them. Usually separate tools are used for writing software and designing hardware. Compilers which can partition software and hardware are rare and cumbersome to use. Also, synthesis tools are required to program the reconfigurable logic. Therefore, the overall design and implementation cycle is difficult and demands care from the designers and implementers of the system.

Furthermore, when moving towards multi applications, multi tasking scenarios, it is even more difficult for the designers to deal with such systems as the exact configuration of the system is not known at design time. These requirements necessitate the existence of a runtime system which is responsible for operating the system and performs the resource management [1]. The resource management by itself is a very complicated task and is dependent on the available information for decision making [2].

One important tool required to achieve this is a runtime profiler which can give vital statistics about the code running on the reconfigurable computer. Those statistics can then be used by the runtime system to decide which parts of the code need to be translated into hardware.

In this paper, we present a runtime profiler, which is intended to be running concurrently with the applications in the actual application execution time. Therefore, one key difference between such a profiler and traditional design time profilers such as *gprof* is that it has to be very low overhead. Furthermore, the information collected by such a runtime profiler needs to be stored in special data structures in such a way that storing and retrieving them can be performed very fast. The major contribution of this paper is therefore proposing a very light weighted runtime profiler with a well defined interface which enables it to be integrated in any runtime system.

The rest of the paper is organized as follows. In Section II, we give a brief overview over the background and motivations behind the proposed profiler. Section III presents the design and implementation of the proposed runtime profiler followed by the section IV, which shows the results of the empirical evaluation. Finally, the conclusion of the paper is given in section V.

II. Motivation and Background Overview

The MOLEN runtime environment [3] is intended to completely virtualize the underlying hardware and thus relieve the program developers from the difficulties of hardware design issues. This runtime system deals with the mapping of the computation intensive parts of the program code to the reconfigurable fabric. The block diagram of MOLEN's runtime system is shown in figure 1. The *scheduler* reads statistics from the profiler about the profiled kernels and decides which to allocate the hardware. In this paper, by kernel we mean computation intensive functions which can be mapped into hardware. The *scheduler* uses the services of the *Transformer* to replace the software implementation of a kernel to its hardware implementation. The *Kernel Library* contains a precompiled set of tasks, saved in form of metadata and containing information such as execution time, power consumption and configuration latency among others. In this paper, we focus on the profiler component which has a very important role in the runtime system.

Our profiler has to communicate with the scheduler at runtime and therefore it needs to use data structures which allow for fast writing and reading of the profiled statistics. Furthermore, it has to be able to continuously profile the applications behavior. This is a fundamental difference between our profiler and profilers used in virtual machines like Java. In Virtual machines, once a function is optimized,

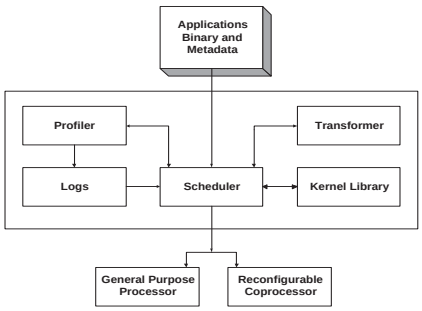

Figure 1. MOLEN's Runtime Environment

it no longer needs to be profiled. On the other hand, in our case, there is limited available hardware, and kernels may be swapped several times. Therefore, they need to be continuously profiled.

Until now, the trend for profiling reconfigurable systems was to use hardware based profilers, but our profiler takes mature instrumentation and sampling techniques from existing off line profilers and apply them for profiling reconfigurable systems. This solves the portability problem of the hardware based profilers. Moreover, using a software based profiler we can easily work with virtual addresses of the operating systems which is a must when profiling software functions.

Table I summarizes different techniques used in the literature for profiling. It covers the instrumentation based profilers, sampling based profilers and hardware based profilers. In this table, we discuss different methods used in each type of profilers as well as the advantages and disadvantages for them. We also included some example for each type.

We have to work with executable binaries and therefore, we cannot perform compile or link time code injection like what *GProf* or *ATOM* do. We need to either modify the executable binary, so that it can contain and use an instrumentation code or use a hardware approach as used by the *frequency loop detection profiler* and *DAProf*. Although the hardware approach has a very low overhead, it has some disadvantages. One obvious disadvantage is that it is not very portable. A hardware design made for one system might not fit into another one. The second disadvantage is that we need to take care of the virtual addresses in the system. At the hardware level we only see the physical addresses. In cases such as *frequency loop detection profiler* which only deals with small loops that can be seen from the instruction bus, they don't require knowing the virtual addresses. On the other hand, in our case, in which we deal with software implementations, we require to know the virtual addresses of the instructions in the programs.

To implement the instrumentation part of the profiler in software, one possible approach is to do a native binary code interpretation and JIT compiling, like *Pin* and *Dynamo*. Although this approach is very useful to create versatile profiling tools like *Pin*, it has some disadvantages. The first of which is that it is slow. We created a *Pin* tool which just counts the number of time each function is called in an application and observed that the overhead was never less than 20%. Another disadvantage of this approach is that it is very complicated and time consuming to write a native binary code interpreter and JIT compiler for any processor.

In our profiler, we choose to replace the function prologue with an unconditional jump to the instrumentation code and have the instrumentation code contain the removed prologue as well as an unconditional jump back to the remaining part of the instrumented function. This approach has some advantages. Firstly, with this approach, low overhead profiling can be achieved by using an efficient instrumentation code. Secondly, this approach is much less complicated than the native binary code interpretation approach. Lastly, this

Table I
COMPARISON OF DIFFERENT TYPES OF PROFILERS

Profiling Technique	Method used	Advantages	Disadvantages	Examples
Instrumentation	Instrumentation Code Insertion at Compile and Link time	Platform Independent	Cannot work with executable binaries	GProf [4]
	Instrumentation Code Insertion during linking of object files	Easy to port	Cannot work with executable binaries	ATOM [5]
	Interpretation of Native Binary Code and JIT Compilation	Instrumentation Code can be inserted anywhere in the code	Has relatively large overhead. Very complicated and time consuming to create such a tool.	Pin [6], Dynamo [7]
	Modification of function prologues to jump to an Instrumentation Code	Has relatively low overhead. Easy to implement and port.	Instrumentation Code cannot be called from anywhere in the code. A function prologue must be of a certain minimum size, so that it can be replaced with a jump to the Instrumentation Code.	Detours [8] and IgProf [9]
Sampling	Using Timer Interrupts	Has low overhead and is non-intrusive	Can only give statistical approximation about time spent by different parts of the code	GProf and [10]
	Using Timer Interrupts and Hardware Performance Counters	Has low overhead and is non-intrusive. Gives additional information like the parts of code causing more cache misses, pipeline stalls etc.	Same as above	OProfile [11]
	Software Based	Easy to port	Extra code has to be injected	Arnold and Ryder [12] and Profiler for IBM TestaRossa [13]
Hardware Based	Using custom-designed hardware	Has very low overhead and is non-intrusive	It is difficult to port a design to another system. Consideration has to be given to handle virtual addresses.	Frequency Loop Detection Profiler [14] and DAProf [15]

technique is very portable, because one has only to deal with function prologues and not the rest of the code inside the functions.

One important issue in the instrumentation code is the way it saves the collected information. As the interaction between the scheduler and profiler is in real time, using normal files is not an option and the data has to be kept in the memory. For this purpose different techniques can be used. For example one can use a */proc* file to save the information. In our profiler, we propose the use of a shared memory and double buffering mechanism to store and read profiled data. The shared memory is shared between the profiler and the scheduler. Using this technique, we have been able to lower the overhead of the profiler to as low as less than 1% for typical applications.

Another contribution of our profiler is in how we use a daemon that runs continuously and injects instrumentation code to applications without any input from the user. The OS kernel is modified so that it sends a signal to the daemon whenever a new program is loaded for execution. On receiving that signal, the daemon injects instrumentation code into the application. To inject code, the daemon uses the *Injector* utility. After the instrumentation code is injected into the application, the application can perform self-instrumentation, that is, it automatically update function call counts in shared memory on entering functions. In this way, our profiler is able to profile multiple programs run by multiple users, without their input and without them noticing it. Another novel technique in our profiler is how we used an off line program, known as the *Extractor*, to extract vital information from a program, so that the code injection takes minimal time. For each program, the *Extractor* also creates a file which contains function ranges, so that the daemon can know which function was executing when a sample at a timer interrupt was taken. This is done by using program counter value at time of the interrupt and process ID of the interrupted process. In this way, the daemon can calculate approximate time spent by functions.

III. Design And Implementation

As explained before, our runtime system is intended to make a completely virtualized and transparent hardware layer available to program developers. In such a system, the programs are developed to be executed on the General Purpose Processor(GPP) and therefore the program developers are not bothered with complex hardware design issues. It is thus the runtime system's responsibility to map those parts of the application which can be accelerated into the hardware. Our profiler has to be able to profile the applications on the GPP in the first place and when a kernel is mapped into the FPGA, it has to be able to continue the profiling of the kernel on the FPGA. As a result, our profiling mechanism has two aspects; the profiling on the GPP, and the profiling on the reconfigurable fabric.

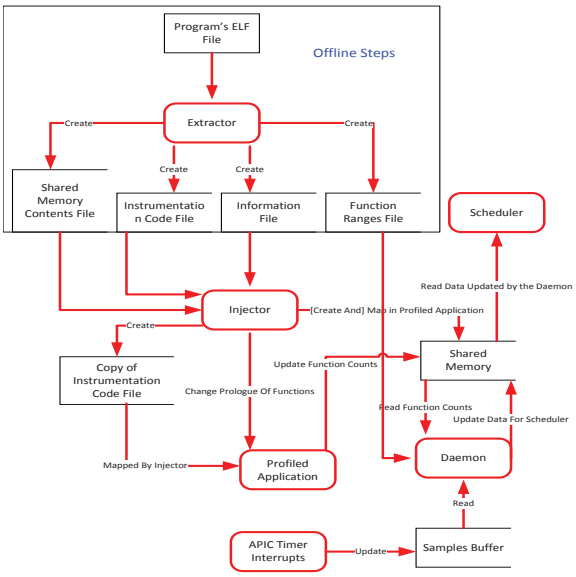

Figure 2. Interaction of different parts of our profiler with the *profiled application* and the *scheduler*

As on the reconfigurable side, we use the MOLEN hardware organization and MOLEN programming paradigm, we employ the MOLEN runtime primitives [16], to perform the profiling. This will be discussed in more detail later in this section.

On the GPP side, our profiler runs on the Linux operating system. Its task is to keep statistics of the running kernels on a general purpose processor and present that information to the *Scheduler*. Our profiler performs both Instrumentation and Sampling profiling. The current implementation has been done and tested successfully on machines with x86 processors, both single-core and multi-core.

Through Instrumentation profiling, we want to record the number of times different functions have been called. To instrument programs, we need some mechanism to inject code into programs. In this way, our code injector replace the prologue of each function that has to be profiled with an unconditional jump to the instrumentation code and the instrumentation code contains a jump back to the rest of the profiled function besides the prologue of the profiled function.

We are not only interested in the number of times different functions are called, but also approximate time spent per function. For that purpose, we perform sampling.

Our profiler consists of several different parts. The interaction between those parts is shown in figure 2. The *Extractor* utility is used to create instrumentation code file besides others, from an executable *elf* file. The *Injector* utility is used to inject instrumentation code and map *Shared Memory* into the address space of the profiled program. The *Injector* utility uses the *Ptrace* API in Linux to inject and modify code at runtime. The local APIC Timer interrupts update the

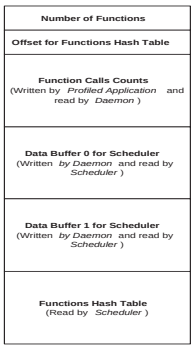

Figure 3. Contents of Shared Memory

Samples Buffer. Note that in case of processors other than x86, we can use their own local timer interrupts to update the *Samples Buffer*. Finally, the *Daemon* combines information from different places in a form that can be quickly and easily read by the *Scheduler*. The purpose of the *Shared Memory* is to keep the *function calls counts* of profiled functions as well as information to be read by the *Scheduler*. The reason we have employed the *Shared Memory* technique is because it is the fastest possible method to communicate data.

One very important part of the profiler is the shared memory structure and the access mechanism to it. This shared memory is in fact the interface of our profiler with the other components of our runtime system. Knowing this interface, it is very trivial to incorporate our profiler to any other runtime system as well. Figure 3 shows a schematic view of the shared memory. The middle portion of the *Shared Memory* is used to present data to the *Scheduler*. That data contains the difference of function calls counts and functions samples counts for a given span time. We also used a double buffering mechanism through which we make sure that the scheduler does not read inconsistent data, i.e., at the same time that the daemon is writing to the shared memory. At the bottom of the shared memory is a hash table which is indexed by a function name or ID and points out to the corresponding location of the collected data in the middle portion.

When a kernel is mapped to the hardware, its profiling has to be continued in order to have a valid (updated) status of the systems at each point in time. Within the MOLEN programming paradigm, the mapping of the kernels to the reconfigurable fabric is being done by the MOLEN SET and EXECUTE APIs [16]. This way, the profiling can be delegated to these APIs when the hardware execution is happening. Each kernel intended to be executed on the hardware should be invoked by the EXECUTE API from the runtime system. Therefore, in the execute phase we exactly know which kernel is going to be executed and as a result the EXECUTE can update the corresponding values in the shared memory. This mechanism guarantees that function call counts updates in case of hardware execution. To update the values in the shared memory, we use the same code as we use in the injected codes in the instrumentation profiling.

Table II
INSTRUMENTATION OVERHEAD (SECS)

	Normal	**Profiling**	**overhead**
multiply	9.681	10.092	4.25%
coremark	12.496	12.654	1.26%
tcf	7.083	7.089	$< 1\%$
h264enc	40.774	40.865	$< 1\%$
minisat2	29.004	29.155	$< 1\%$

IV. PERFORMANCE EVALUATION

To test the performance of our profiler, we have used benchmarks from different areas. One of this is PC version of *tcf*, which is a Stationary Noise Filter used in hArtes [17] demonstration. We used a snd file of size 19 MB as the input. Then we have *minisat2* [18], which is an industrial scale SAT solver. Note that we have removed the randomness part in *minisat2*, so that our results do not vary from run to run. The input file for the *minisat* benchmark contained 630 variables and 2280 clauses. Then we have *H264/AVC encoder*, from MediaBench II benchmarks [19], which is an H264 encoder application. For the H264 encoder application, we used an input file of size 5.2 MB and bit rate of 45020 bps. Next, we have *coremark* [20], which is a free synthetic benchmark from *EEMBC* [21]. Finally, we have a benchmark that we created ourselves, known as *multiply*. That program calls a multiply function which just returns product of two numbers, one billion times. Normally, such tiny functions are inlined by the compiler and therefore profiling would not be required for them. However, testing the profiler with such tiny functions gives us an idea of the worst case performance of the Instrumentation Profiling. To avoid inlining of the multiply function, the *multiply* application is compiled with optimizations turned off.

We performed different experiments which are discussed in the following.

A. Instrumentation Overhead

Table II shows the instrumentation overhead of our profiler. From the table we can see that the overhead of our profiler, except for the *multiply* application, is always less than 1.5%. The low overhead for applications other than *multiply* was expected because our profiler only adds three instructions to original functions for profiling. The overhead for the *multiply* application here is relatively large because it repeatedly calls a very tiny function that just returns product of two numbers.

The readings given in table II were calculated by running each of the benchmark ten times and then taking the mean values.

B. Sampling and Daemon Overhead

In table III, we show the results achieved without performing instrumentation. The purpose of not performing instrumentation in this case is to quantify the overhead imposed by sampling and the *Daemon*. The results are compared with those achieved from *OProfile*. The readings

Table III
SAMPLING AND DAEMON OVERHEAD

	Normal	OProfile	Our Profiler
multiply			
Time (secs)	9.681	9.677	9.688
Overhead	-	$< 1\%$	$< 1\%$
coremark			
Time (secs)	12.496	12.510	12.495
Overhead	-	$< 1\%$	$< 1\%$
tcf			
Time (secs)	7.083	7.121	7.126
Overhead (%)	-	$< 1\%$	$< 1\%$
h264enc			
Time (secs)	40.774	40.838	41.034
Overhead	-	$< 1\%$	$< 1\%$
minisat2			
Time (secs)	29.004	29.037	29.058
Overhead	-	$< 1\%$	$< 1\%$

given in table III are means of 10 readings. From that table, we can see that the overhead for both *OProfile* and our profiler is negligible. It has to be noted here that we only used the *timer interrupt event* for *OProfile*, so as to make it functionally equal to our profiler.

C. Sampling Accuracy

In this part, we checked the accuracy of the sampling part of our profiler by comparing it with gprof [4]. Here we used the same benchmark applications that we used in the previous section, i.e., those which take more than 10 seconds to execute. We ran each program five times, both with our profiler and with *gprof*, so that we could get the mean values . The results are given in table IV. In this table, means of percentages of total time spent for the functions which took the most time according to *gprof* are given. The functions are sorted by percentages of total time spent, given by *gprof*. From the table, it can be seen that the mean values for both our profiler and *gprof* are almost the same. This was expected as we are using the same technique as *gprof*. The only difference is that we take our samples through the local APIC timer interrupts, so that we can take samples for multi-cores, while *gprof* uses the kernel timer interrupt and therefore cannot perform sampling for multi-cores. Since the default Linux kernel timer interrupt occurs at the rate of 100 per second, we also set the frequency of the local APIC timers interrupt to 100 per second for fair comparison.

D. Overall Overhead

In the first part of this experiment, we tested our benchmark applications with all parts of the profiler working. The means of ten readings are shown in table V. The results are as expected, that is all applications other than *the multiply* application have overhead of less than 1.5%. Moreover, overall overhead for all application is almost the same as that for instrumentation overhead, thus reinforcing the fact that sampling and the *daemon* have very low overheads.

In the second part of the experiment, we ran all the benchmark applications simultaneously with all parts of the

Table IV
SAMPLING ACCURACY OF OUR PROFILER

Function	gprof (%)	Our Profiler (%)
coremark		
crcu8	31.77	31.13
core_state_transition	30.05	31.20
core_bench_list	13.85	15.14
matrix_mul_matrix_bitextract	5.48	5.55
minisat2		
Solver::propagate	74.87	75.05
Solver::analyze	13.56	13.53
Solver::litRedundant	4.45	4.43
Solver::cancelUntil	2.85	2.92
h264enc		
SetupFastFullPelSearch	33.59	32.63
dct_luma	11.17	10.60
biari_encode_symbol	7.33	7.30
SetupLargerBlocks	3.83	3.42

Table V
OVERALL OVERHEAD OF OUR PROFILER (SECS)

	Normal	Our Profiler	Overhead
Single Application Execution			
multiply	9.681	10.112	4.45%
coremark	12.496	12.655	1.27%
tcf	7.083	7.088	$< 1\%$
h264enc	40.774	41.158	$< 1\%$
minisat2	29.099	29.040	$< 1\%$
Multiple Applications Execution			
Five benchmarks	92.32	93.04	$< 1\%$

profiler working. We repeated the experiment five times and took the mean values which are shown in bottom of table V. The results show that the profiling overhead is less than 1%.

Our results shows that our profiling system is at par with *Dynamo*, which has overhead of less that 1.5% and better than the profiler presented in [12] which has average overhead of 3%. It has to be noted though that pieces of code which are optimized by Dynamo are never profiled again, while our profiler has to continuously profile all the functions.

E. Percentage of Profilable functions

Our profiler replaces prologues of to be profiled functions with a jump instruction. For that purpose, a function's prologue must be at least 5 bytes because the jump instruction in an x86 consumes 5 bytes. Most of the functions do have at least 5 bytes of prologue, but some functions, usually of very small size, do not. By prologue instructions, we mean instructions which prepare the stack and registers for use within a function and not any instruction that is used afterwards. In table VI, we have shown the number of function which are profilable for different applications. We have also listed the optimization levels used to compile those applications, the purpose of which is to see if optimizations make it any harder to find profilable functions. Except for *coremark*, all applications have more than 90% of profilable functions and both *h264enc* and *minisat2* are using high level of optimizations. The reason that *coremark* has only 72.5% of profilable functions is because there are many functions of very small sizes.

We have to only use prologue and that is why we have the limitation on the size of it. One can say if the prologue

Table VI
PERCENTAGE OF PROFILABLE FUNCTIONS

Program	Opt Level	Total Functions	Profilable Functions
tcf	-O0	59	59 (100%)
h264enc	-O2	591	560 (94.8%)
minisat2	-O3	56	52 (92.9%)
coremark	-O2	40	29 (72.5%)

is too small you can include more instructions from the top of function and consider them as a part of prologue. But, this solution is not feasible because it might happen that an instruction inside a function jumps to some instruction at the top of that function, and if that instruction at the top is replaced by some other instructions, the program might crash or behave differently.

V. CONCLUSION

In this paper, we described the design and implementation of a runtime profiler which can be used as a part of the MOLEN runtime environment. The profiler is a combination of a Sampling profiler and an Instrumentation profiler. We discussed different parts of the profiler namely the *extractor*, *injector*, *sampler*, *shared memory* and *daemon*. Then we showed the overhead of our profiler from different aspects. This is done by showing the overhead on Instrumentation, Code Injection, Sampling and the overall overhead. Besides, we compared the accuracy of our profiler with a popular design time profiler. All the presented results show that our profiler has very low overhead (less than 1.5%) and is as accurate as design time profilers.

ACKNOWLEDGMENT

This research is partially supported by hArtes project (EUIST-035143), Artemisia iFEST project (grant 100203), Artemisia SMECY (grant 100230) and FP7 Reflect (grant 248976).

REFERENCES

[1] M. Sabeghi and K. Bertels, "Toward a runtime system for reconfigurable computers: A virtualization approach," in *Design, Automation and Test in Europe (DATE09)*, April 2009.

[2] M. Sabeghi, V. Sima, and K. Bertels, "Compiler assisted runtime task scheduling on a reconfigurable computer," in *19th International Conference on Field Programmable Logic and Applications (FPL09)*, August 2009.

[3] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers, Vol 53, No. 11*, pp. 1363–1375, 2004.

[4] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," *ACM SIGPLAN Notices, Vol 39, No. 4*, pp. 49–57, 2004.

[5] A. Eustace and A. Srivastava, "Atom: a flexible interface for building high performance program analysis tools," in *Proceedings of the USENIX 1995 Technical Conference*. Berkeley, CA, USA: USENIX Association, 1995, pp. 25–25.

[6] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 190–200.

[7] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *ACM SIGPLAN Notices*, 2000, pp. 1–12.

[8] G. Hunt, , G. Hunt, and D. Brubacher, "Detours: Binary interception of win32 functions," in *In Proceedings of the 3rd USENIX Windows NT Symposium*, 1998, pp. 135–143.

[9] G. Eulisse and L. A. Tuura, "Igprof profiling tool," *Computing in High Energy Physics and Nuclear Physics*, p. 665, 2004.

[10] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, "A dynamic optimization framework for a java just-in-time compiler," in *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2001, pp. 180–195.

[11] http://oprofile.sourceforge.net/, 2010.

[12] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code," *ACM SIGPLAN Notices, Vol 36, No. 5*, pp. 168–179, 2001.

[13] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley, "Experiences with multi-threading and dynamic class loading in a java just-in-time compiler," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 87–97.

[14] A. Ross and F. Vahid, "Frequent loop detection using efficient nonintrusive on-chip hardware," *IEEE Transactions on Computers, Vol 54, No. 10*, pp. 1203–1215, 2005.

[15] A. Nair and R. Lysecky, "Non-intrusive dynamic application profiler for detailed loop execution characterization," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA: ACM, 2008, pp. 23–30.

[16] M. Sabeghi and K. Bertels, "Interfacing operating systems and polymorphic computing platforms based on the molen programming paradigm," in *Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2010.

[17] http://www.hartes.org/, 2010.

[18] N. Een and N. Sorensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing*, 2004, pp. 333–336. [Online]. Available: http://www.springerlink.com/content/x9uavq4vpvqntt23

[19] http://euler.slu.edu/~fritts/mediabench/mb2/, 2010.

[20] http://www.coremark.org/, 2010.

[21] http://www.eembc.org/, 2010.