

# Extending a Light-weight Runtime System by Dynamic Instrumentation for Performance Evaluation

Mario Kicherer, Fabian Nowak, Rainer Buchty, Wolfgang Karl

Karlsruhe Institute of Technology  
Chair for Computer Architecture  
76128 Karlsruhe, Germany  
E-Mail: {kicherer|nowak|buchty|karl}@ira.uka.de

## Abstract

Increasing complexity of current and future systems poses a new challenge for software engineers. In a previous work we presented a light-weight runtime system for abstraction of heterogeneous parallel systems. This runtime system adapts an application to the current system state in order to improve the utilization of the available resources. For online evaluation of such adaptations, we propose a versatile instrumentation mechanism that can be used by the runtime system. We show that this technique introduces only minor overhead and we compare it to the Dyninst mechanism.

## 1 Introduction and Motivation

While the automatic performance gain through higher frequencies and superscalar processors has come to an end [12, 15, 6], the number of transistors on a chip increases even further, as predicted by G. Moore [8]. The answer to the question of how to use them reasonable is already visible these days, as almost all major vendors only sell CPUs with two or more cores per processor for the desktop and high-performance market. While today's available processors have at most eight cores, the trend for the upcoming years is already visible in terms of the Intel Larrabee [13] or, more recent, the Intel Single-chip Cloud Computer (SCC) with its 48 x86 cores.

Besides the increasing number of cores, people predict that the diversity of computational units in a system will rise, too. While it is standard to have a powerful GPU in a commodity system – the newest generation already passes the 2 TFLOPS mark – there are also numerous high-performance systems that have additional co-processors like GPUs, Cell processor units or FPGA chips. In future systems, there might even be heterogeneous units on a die, as proposed in [4, 18] or by the creators of the MOLEN Polymorphic Processor [17]. The benefit of such architectures is that in theory there is a specialized core available for every problem with, depending on the goal, a high throughput or minimal energy consumption per calculation, for example. The problem that comes up is how to use these resources efficiently. Two subproblems that need to be solved are finding a suitable computation resource and easing the programming of applications for such systems.

In [1], we introduced the DLS runtime system with focus on a light-weight design that tackles these issues. Its key functionality is the ability to reroute function calls al-

most transparently for the software engineer on the behalf of a central control instance. Through the rerouting, it is possible to exchange implementations. These implementations can have different characteristics, for example, they can use different resources for computation or provide certain features like accuracy or adhere to deadlines. In [9] and [10], we proposed a mechanism based on so-called attributes that can be used to define the requirements of an application, for example minimum accuracy, and the characteristics of an implementation, for example provided accuracy. Using the ELF format, these attributes can be stored in the same binary file as the compiled code itself and do not interfere with legacy operating systems. This way it is possible for a control instance to calculate an assignment of task to implementation that aligns the system with the chosen optimization goal.

In case an adequate attribute classification does not exist for a specific implementation, there is currently no proper way to rate the potential benefit or loss after its execution. If an application shall be optimized for speed, for example, the most promising input value for the rating process would be the overall execution time. If we have a complex application, containing other exchangeable implementations or indeterminable factors, or even several applications that execute in parallel, this value might be distorted and thus will not be adequate. Other factors that might interfere with performance rating could be another running application that is out of control of the runtime system and has an unpredictable influence on shared resources, or implementations that distribute the workload over a network to other nodes, for example using MPI. All these cases can have a major influence on the efficiency of the scheduling and mapping of the control daemon. For this reason, a fine-grained performance evaluation is nec-

essary to estimate the benefit of a specific implementation. We therefore propose a light-weight instrumentation mechanism that is able to survey a chosen implementation for individual threads. The results can be stored in the already existing data structures of the runtime system and can be read by the control daemon for mapping and scheduling decisions.

This paper is structured as follows: In Section 2, we present related work in the area of instrumentation. Section 3 explains our concept of dynamic instrumentation and its integration in the runtime system. We discuss the introduced execution overhead of our approach in Section 4 and compare it with another solution, before we draw the final conclusions and give an outlook over our future work in Section 5.

## 2 Related Work

Instrumentation is a topic with a long history and many solutions for different areas and architectures. Some groups, such as ATOM [14] and EEL[5], establish instrumentation through static modifications. As this adds a fixed overhead, it is not usable for our purposes. There is also a number of solutions for individual languages like InsECT [2] that are specialized on instrumenting Java applications using byte-code transformation.

In [16], they present a method for code coverage testing and use a dynamic injection technique called Paradyn, introduced in [3], to monitor which part of the code is executed. In detail, an external application can inject binary code into the to-be-monitored program at runtime and can remove it again. In contrast to our approach, the instructions for instrumentation must be defined using the so-called Dyninst-API [11], that originates from the Paradyn project. In our approach, theoretically any regular function can be used as pre- and post-call function. A benefit of their approach is that they can instrument arbitrary functions.

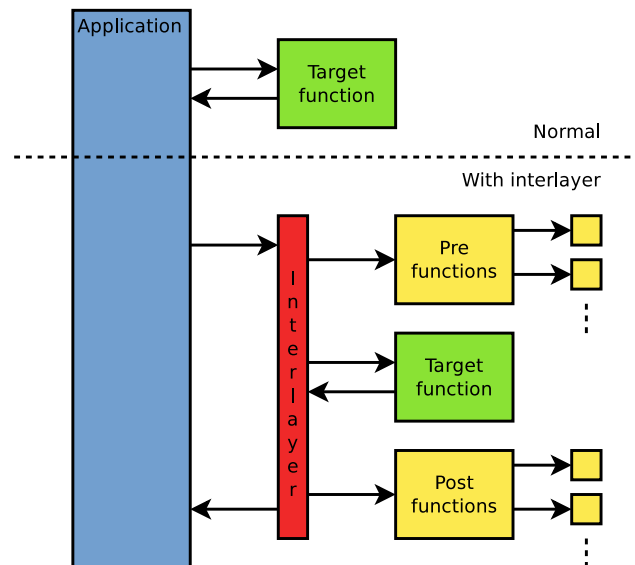
PIN [7] is another solution that is capable of dynamic instrumentation. It is developed by Intel and supports, among others, 32- and 64-bit architectures on the major operating systems. PIN achieves instrumentation through dynamic binary rewriting. It consists of a virtual machine with a just-in-time compiler, a code cache, an emulation- and a dispatcher unit. To instrument a running application, it attaches the process and instruments every path through recompilation in order to keep control over it. A benefit is that through the recompilation process, optimizations can also be applied that reduce the overhead introduced by the code for the instrumentation.

Both solutions for dynamic instrumentation provide the functionality to instrument an application at any point of the execution. These pervasive methods open many options in the area of instrumentation, but are too heavy-weight for simple approaches like the selective measurement of time consumption. Also, for our purpose, it is

important that the generated data is easily available for the control daemon. While both solutions require a separate mechanism to relay the gained information, we achieve this by using the structures of the DLS runtime system that are already existent in the application and readable through the proc filesystem of the kernel.

## 3 Dynamic Instrumentation

In our design, instrumentation is established through the introduction of the so-called *interlayer* in the DLS runtime system. The interlayer is placed between the caller and the callee function and enables the ability to call arbitrary functions prior to and after the callee function as depicted in **Fig. 1**. These functions can execute any code to gain any information from the working data, system, or process itself. To relay the collected values to the external control daemon, the interlayer also provides the address of the management structure of the corresponding proxy. A proxy is the key functionality of the DLS runtime system, introduced in [1]. In fact, it is a function pointer that can be modified from an external process through the kernel module of the runtime system. Every management structure of a proxy contains special fields to store data and is accessible through the kernel over the proc filesystem. The control daemon can read the output and utilize this information for its scheduling and mapping decisions.



**Figure 1:** Normal and interlayer program flow

Using the mechanisms of the DLS runtime system, it is possible to dynamically activate the interlayer and the individual functions. In fact, the proxy is modified in such a way that it does not point to the address of the target function, but to the address of the interlayer. So, the interlayer and the instrumentation functions can be activated and disabled at any time from the application itself or through the kernel interface without modifications to the binary code.

In general, the interlayer is not limited to the DLS runtime system. With minor adjustments this mechanism can be used in most applications or operating systems. For example, one can also replace direct with indirect calls in the application and activate instrumentation by using a debugging mechanism like ptrace to modify the address in a way that it points to the interlayer.

The overall problem that arises is that the target function has to get the right arguments and the return value has to be passed back although other functions are called before and after. The interlayer solves this by specific stack modifications. Essentially, it copies the caller stack frame on top of the stack and calls the other functions. A view of this process on the stack is illustrated in Fig. 2. The detailed algorithm is explained in the following.

The first task is to calculate the boundaries of the stack frame and the source and destination addresses of the copy procedure. The current stack frame starts at the location stored in `%ebp`, so we save it as source address in `%esi` (line 1). As the old base pointer and return address are not required in the new stack frame, we add a value of 8 (line 2). In line 3, we copy the old base pointer as upper boundary to the `%ecx` register. In the next step, we subtract the lower boundary address to get the size of the stack frame. In line 5, this value is saved in `%ebx` for later use. The copy instruction calculates in word units, so we rotate the size right by 2 digits in line 6, what equals a division by 4. In the next step, the stack pointer is moved so we have free place for our new stack frame. In line 8, we copy the stack pointer as destination address to the `%edi` register. The instructions in line 9 and 10 copy `%ecx` words from `%esi` to `%edi`.

```

mov %ebp, %esi      1
add $8, %esi        2
mov (%ebp), %ecx    3
sub %esi, %ecx      4
mov %ecx, %ebx      5
ror $2, %ecx         6
sub %ebx, %esp       7
mov %esp, %edi       8
cld                  9
rep movsl            10

```

As the new stack frame is ready, we can call the `dls_pre_call` function, that in turn calls the registered instrumentation procedures. Before, we push the address of the proxy management structure as first argument. After the call, we remove it by incrementing the stack pointer.

```

push proxy_mngmnt_structure 11
call dls_pre_call           12
add $4, %esp                13

```

Now we call the function at the address in `%eax`, because this is the return value of the `dls_pre_call` function that returns the address of the target function. We don't call the proxy itself again, as it might be necessary that it still points to the interlayer function, for example in case

of continuous time measuring, instead of pointing directly to the target function.

```
call *%eax           14
```

In the next step, we backup the return value that, depending on the type, is stored in `%eax` and `%edx`, because the callee can use them without caring about the initial value, as defined in the `cdecl` calling convention for x86 code. After this, the `dls_post_call` function is called the same way as the `dls_pre_call` function and we restore the return values.

```

push %eax           15
push %edx           16
push proxy_mngmnt_structure 17
call dls_post_call  18
add $4, %esp        19
pop %edx            20
pop %eax            21

```

In the last step, we have to manually clean up the stack. So we simply add the size of the previously copied frame to the stack frame and program execution can continue in the regular fashion.

```
add %ebx, %esp      22
```

## 4 Evaluation

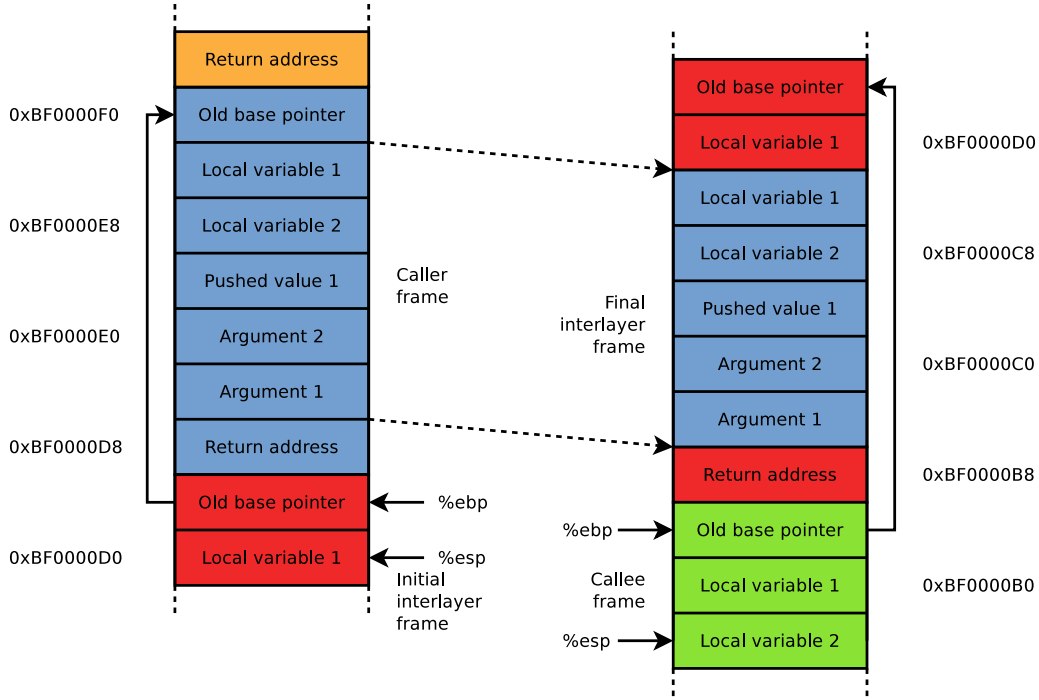
The maxim of the DLS runtime system is a low-overhead design. As we demonstrated in our previous paper [1], the introduction of proxies in an application does not create any measurable overhead. Only the process of changing implementations imposes a minimal overhead, which, due to the rate of such events, is negligible in a real application. In this section, we present the results of our first measurements of the instrumentation and compare these results with Dyninst [11].

For our evaluation, we used an Intel Pentium 4 processor with 3.2 GHz and 1 GB main memory, running Arch-Linux with a 2.6.31-kernel patched for the operation with the DLS runtime system.

	time for 10 <sup>6</sup> iterations
w/o	6 * 10 <sup>6</sup>
Basic instrumentation	57 * 10 <sup>6</sup>
Instrumentation with Time-measuring	2137 * 10 <sup>6</sup>

**Table 1:** Time consumption in ns for normal and instrumented execution and with time-measuring

To measure the overhead of the instrumentation, we have created a stress-test application that comes with a minor amount of actual calculation only, but which is repeated many times. In detail, the application calls a function in a library that simply returns an integer 1,000,000 times. This application was executed thirty times and the runtime was measured using the `CLOCK_THREAD_CPUTIME_ID`



**Figure 2:** Duplication of caller stack frame through the interlayer

clock source of the `clock_gettime` function. In **Table 1** we see that the uninstrumented function call lasts average 6 nanoseconds. If the instrumentation infrastructure is activated, the time consumption raises to 57 nanoseconds. So, 51 ns are required for the procedure presented in Section 3 – essentially for determining the old stack frame and copying it onto the top of the stack. This represents an increase in time consumption of nearly one order of magnitude. If we activate a simple function for instrumentation that takes the timestamp before and after the target function and saves the difference in the management structure, the time consumption increases by another two orders of magnitude. So, in a real-life example, the introduced overhead of only the instrumentation infrastructure would be under 3% in most cases.

	time for $10^6$ iterations
Our approach	$57 * 10^6$
Dyninst start	$132 * 10^6$
Dyninst start&end	$243 * 10^6$

**Table 2:** Time consumption in ns for basic instrumentation using our and the Dyninst approach

For comparison, we also instrumented this application using Dyninst [11]. We used the procedure in the basic example described on page 8 and 9 in the Programming Guide of the Dyninst v6.1 package to instrument the mentioned function. In this example, Dyninst inserts instructions to increase a counter at every code location that will be instrumented. Dyninst only inserts instrumentation instructions at one code location at a time. Therefore, one time,

we measured the overhead for instrumentation solely at the start of a function and a second time, at the start and at the end of a function. As we can see in **Table 2**, the instrumentation using Dyninst lasts 231%, respectively 426%, longer than our approach.

# of arguments	1	4	8	16	24	32	48	64	128
time	57	57	57	65	73	86	104	120	188

**Table 3:** Time consumption in ns for a variable number of 32-bit arguments

The only varying factor in the time consumption of the instrumentation infrastructure is the size of the stack frame. The size is determined by the number of local variables of the caller function, the number of values it pushed onto the stack and the number of required arguments of the function to be called. In order to have various sizes of the stack frame, we measured the time overhead of an increasing number of arguments passed to the target function, which leads to the desired increase in stack frame size. This is feasible because it does not make a difference for our purpose, if a value in the stack frame is a local variable, pushed value or a function parameter. As it can be seen in **Table 3**, there is no significant increase in the overhead until the number of 32-bit arguments exceeds a value of eight. So, the number of arguments has only a negligible impact on the runtime in ordinary applications, as the number of arguments seldom passes the number of 16 or even 8. The far more important factor is the number of local and pushed variables because their number can be very high in complex functions. So, for complex functions and

frequent measurements, an intermediate dummy function with a small stack frame might lower the time overhead in such cases.

## 5 Conclusion and Outlook

To calculate a good scheduling or resource mapping, respectively, the control daemon needs accurate information about the requirements of the applications and about the characteristics of the available implementations. In a previous publication, we presented a method to define both of them as so-called attributes and store them in the respective ELF files. For example, when an implementation depends on system parameters that vary during runtime, the significance of these attributes might not be sufficient. Hence, there is a need for another method to rate the efficiency of the implementation.

For this reason, we proposed a light-weight instrumentation solution that is capable of examining a chosen implementation and integrates tightly into our previously developed dynamic linking system. It is based on slight stack modifications to execute arbitrary functions prior and after the callee function. The intermediate functions can be used to survey the performance of a function in the required fashion, such as execution time or – in a limited way – power consumption.

As we demonstrated in Section 4, the instrumentation itself introduces only a minor overhead, if compared with the Dyninst solution or the overhead created by a simple function that does time measuring. We have shown that the overhead is mainly influenced by the size of the stack frame, that is made up of the local and pushed values of the caller and the number of parameters of the callee function. The presented technique is focused on the DLS runtime system. In [1] we introduced another variant, called GOT-based linking system (GLS), that is specialized on dealing with binaries without access to the source code. In a further step, we will therefore investigate how our instrumentation method can be ported to this second approach to support proprietary software.

## References

- [1] Rainer Buchty, David Kramer, Mario Kicherer, and Wolfgang Karl. A light-weight approach to dynamical runtime linking supporting heterogenous, parallel, and reconfigurable architectures. In *Architecture of Computing Systems, – ARCS 2009*, volume 5455/2009 of *Lecture Notes in Computer Science*, pages 60–71. Springer Berlin / Heidelberg, February 2009.
- [2] Anil Chawla and Alessandro Orso. A generic instrumentation framework for collecting dynamic information. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4, 2004.
- [3] J. K. Hollingsworth, O. Niam, B. P. Miller, Zhichen Xu, M. J. R. Goncalves, and Ling Zheng. Mdl: A language and compiler for dynamic program instrumentation. In *PACT '97: Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, page 201, Washington, DC, USA, 1997. IEEE Computer Society.
- [4] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.
- [5] James R. Larus and Eric Schnarr. Eel: machine-independent executable editing. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 291–300, New York, NY, USA, 1995. ACM.
- [6] James Laudon. Performance/watt: the new server focus. *SIGARCH Comput. Archit. News*, 33(4):5–13, 2005.
- [7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.
- [8] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, Volume 38, Number 8, 19. April 1965.
- [9] Fabian Nowak, Rainer Buchty, and Mario Kicherer. Providing guidance information for application-mapping on heterogeneous parallel systems. In *PARS Newsletter #26, GI/ITG*, December 2009. to appear.
- [10] Fabian Nowak, Mario Kicherer, Rainer Buchty, and Wolfgang Karl. Delivering guidance information in heterogeneous systems. In *ARCS Workshop Proceedings 2010*. to appear.
- [11] University of Maryland. Dyninst API. <http://www.dyninst.org/>.
- [12] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, New York, NY, USA, 1996. ACM.

- [13] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.
- [14] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM.
- [15] Herb Sutter. The free lunch is over, 2005. <http://www.gotw.ca/publications/concurrency-ddj.htm>.
- [16] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96, New York, NY, USA, 2002. ACM.
- [17] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The molen polymorphic processor. *IEEE Trans. Comput.*, 53(11):1363–1375, 2004.
- [18] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Dynamic heterogeneity and the need for multicore virtualization. *SIGOPS Oper. Syst. Rev.*, 43(2):5–14, 2009.