

Opportunities for Concurrent Dynamic Analysis with Explicit Inter-core Communication

Jungwoo Ha Stephen P. Crago

University of Southern California
Information Sciences Institute
{jha,crago}@isi.edu

Abstract

Multicore is now the dominant processor trend, and the number of cores is rapidly increasing. The paradigm shift to multicore forces the redesign of the software stack, which includes dynamic analysis. Dynamic analyses provide rich features to software in various areas, such as debugging, testing, optimization, and security. However, these techniques often suffer from excessive overhead, which make it less practical. Previously, this overhead has been overcome by improved processor performance as each generation gets faster, but the performance requirements of dynamic analyses in the multicore era cannot be fulfilled without redesigning for parallelism.

Scalable design of dynamic analysis is a challenging problem. Not only must the analysis itself be parallel, but the analysis must also be decoupled from the application and run concurrently. A typical method of decoupling the analysis from the application is to send the analysis data from the application to the core that runs the analysis thread via buffering. However, buffering can perturb application cache performance, and the cache coherence protocol may not be efficient, or even implemented, with large numbers of cores in the future.

This paper presents our initial effort to explore the hardware design space and software approach that will alleviate the scalability problem for dynamic analysis on multicore. We choose to make use of explicit inter-core communication that is already available in a real processor, the TILE64 processor and evaluate the opportunity for scalable dynamic analyses. We provide our model and implement concurrent call graph profiling as a case study. Our evaluation shows that pure communication overhead from the application point of view is as low as 1%. We expect that our work will help design scalable dynamic analyses and will influence the design of future many-core processors.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Run-time environments

General Terms Experimentation, Performance

Keywords Dynamic analysis, Inter-core communication, Concurrency, Instrumentation

1. Introduction

The exponential improvement in single processor performance has recently come to an end, mainly because clock frequency has reached its limit due to power constraints. Thus, processor manufacturers are choosing to place multiple cores into a single chip, which can improve performance given parallel software. This paradigm shift to multicores require scalable parallel applications that execute tasks on each core, otherwise the additional cores are worthless.

Meanwhile, dynamic analysis techniques have become popular in many areas, such as debugging [6, 11, 14], performance optimization [3], security [12], and software support [8]. Dynamic analysis is usually performed by instrumented code, which adds overhead to the application. When single processor performance was rapidly improving, the overhead could be overcome as single processors became faster. Now that the microprocessor industry has shifted to multicore, single core performance is no longer improving, and dynamic analyses must be carefully designed for scalability to enable the benefits of analysis with reasonable performance overhead.

Since every application has sequential parts, it is inevitable that some cores are under-utilized, and this under-utilization provides opportunities for dynamic analysis to use the extra cycles to do more sophisticated analysis or to increase accuracy without sacrificing performance. Previously, a typical method of keeping overhead low is sampling, which decreases overhead but sacrifices accuracy. While sampling is successful in some areas [1, 2, 7, 10], not every dynamic analysis method can apply sampling [8], and it is always better to achieve higher accuracy if it does not introduce higher overhead.

To achieve scalability for a dynamic analysis system, analysis must be *parallel*, i.e., the analysis itself must run in parallel, and *concurrent*, i.e., analysis runs simultaneously with the application. While many researchers focus on improving scalability of applications and dynamic analysis, it is often neglected that the interaction between the application and analysis threads must also be scalable. Ideally, application and analysis is decoupled, and the application only sends necessary data to the analysis thread. By quickly off-loading the data from the critical path, application overhead will be kept low.

Ha et al. recently introduced a concurrent dynamic analysis framework that minimizes application overhead for transferring data from application to analysis threads [9]. The framework takes advantage of modern cache coherence protocols and hardware prefetchers, which hide the cache-miss penalty caused by using per-thread buffering. However, cache coherence can be inefficient for producer-consumer style communication, and the inefficiency may get worse as the number of cores increase [4]. Many researchers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASTE'10, June 5–6, 2010, Toronto, Ontario, Canada.

Copyright © 2010 ACM 978-1-4503-0082-7/10/06...\$10.00

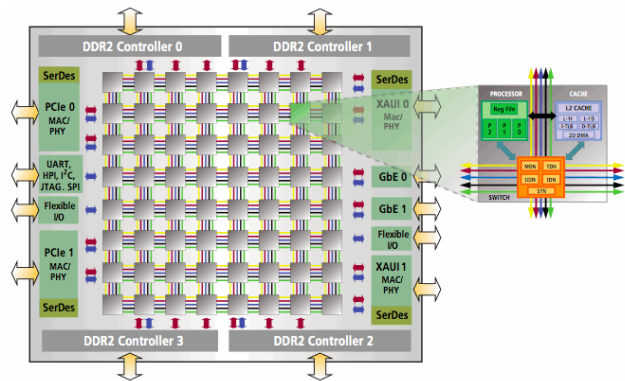


Figure 1. TILE64™ architecture overview

have sought hardware support to reduce the overhead of dynamic analyses [11, 13, 14], but many suggested hardware changes are not generic and are less likely to be adopted in production hardware.

Recently, Tiler released the TILE64 processor, which has 64 cores on a single chip, placed in an 8 by 8 mesh structure. The TILE64 architecture has two explicit inter-core communication networks, the *static network (STN)* and *user dynamic network (UDN)*. A user application can transmit and receive data from these networks via special-purpose registers. Hypothetically, the overhead of writing to a special purpose register is lower than writing to a buffer. By utilizing these explicit interconnect networks, analysis data can be transferred to a remote core without polluting the application's local cache, and dynamic analysis can be performed at a remote core with minimal interaction with the application. Applications incur overhead only from collecting and shipping data to the network.

In this paper, we present a preliminary model for scalable dynamic analysis that utilizes explicit inter-core communication. The purpose of the work is to leverage hardware features on current and future many-core systems for dynamic analysis and to characterize different types of hardware support that benefit scalable dynamic analyses. We implement call graph profiling as a reference, and our result shows that communication overhead of using explicit inter-core communication is as low as 1%.

Upon the successful completion of this research, our work will be applicable to various areas that use runtime analysis, such as parallel debugging, security monitoring, software support, and automatic testing. By showing the usefulness of the interconnect networks in support of dynamic analysis, we hope that our work can influence future multicore design and analysis systems.

2. Tile64 Architecture Overview

The TILE64 from Tiler is the first commercial, general-purpose many-core microprocessor. TILE64, which has been succeeded by the TILEPro64™, has 64 cores on a single die. Each TILE64 core executes a 3-way VLIW RISC instruction set, with SIMD operations for 16-bit and 8-bit operations and in-order execution. Each core has a memory management unit for virtual memory support and executes an independent control thread or process.

There are two levels of local memory associated with each core. The first level of local memory associated with each core can be configured as a cache or scratchpad memory. The second level of local memory functions as a local L2 cache, and can be accessed from remote cores as a third level of cache. The processor has four DDR2, two XAUI, two four-lane PCIe, and two GbE MAC interfaces, as well as serial and programmable interfaces.

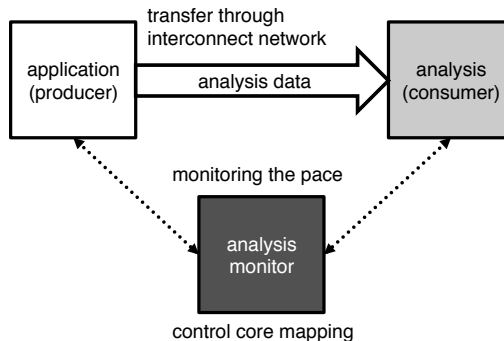


Figure 2. Overview of dynamic analysis framework

The cores of the TILE64 are interconnected by a set of 2D-mesh networks collectively called iMesh™. iMesh is a set of five inter-core networks, each of which perform different functions. Routing through intermediate tiles does not disrupt intermediate tile processors (cores). The Memory Dynamic Network, MDN, routes data between cores and between cores and main memory and is accessible only through the caches.. The Tile Dynamic Network, TDN, routes memory data between tiles and is also accessible only through caches. The I/O Dynamic Network, IDN, routes data from cores to I/O interfaces and is accessible from the cores from operating system code. The last two networks are accessible by user-level code. The User Dynamic Network, UDN, and the Static Network, STN, are both accessible via a register-level FIFO interface from user code directly via assembly code or through a C library. The UDN requires a header for each packet that specifies the destination core. The STN requires routes to be pre-programmed, which eliminates the need for the header, at the cost of less flexible (static) routing.

The TILE64 processor runs C and C++ programs on an implementation of SMP Linux 2.6. For programs or tasks that cannot tolerate the interrupts of a general-purpose operating system, zero-overhead Linux and a bare-metal environment are also available. The cores can be divided into rectangular partitions to provide protection between independent programs. Both multi-processing and multi-threading are supported, and tasks can be pinned to specific cores to prevent process or thread migration, which may be desired for performance reasons and is also necessary when using the user-accessible networks. Packets on the dynamic network can be from 1 to 128 32-bit words, and each packet has one header word. Data in the static network is routed on a word-by-word basis.

3. Concurrent Dynamic Analysis Model for Many-core Systems

Figure 2 shows an overview of our concurrent dynamic analysis model. We model dynamic analysis systems in three parts: *application*, *analysis*, and *analysis monitor*. The application runs with instrumented code which only transfers analysis data to an analysis thread through the interconnect network. The analysis threads read data from the network and perform the analysis. The analysis monitor is a helper thread that controls resources for optimal performance.

The performance goal of the system is to execute the application while minimizing slowdown. However, this does not imply that the application is prioritized during resource scheduling. If the analysis thread does not keep pace with the application, i.e., the application produces data faster than the analysis threads can consume it, more cores need to be delegated to the analysis thread.

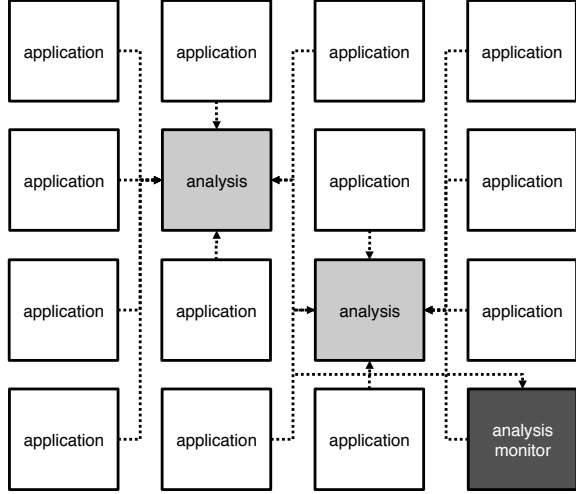


Figure 3. Dynamic tile mapping for application and analysis threads

Benchmark	# of instrumented calls	# of instrumented calls / Mcycles
bodytrack	8984k	1278
blackscholes	8794k	1317
fluidanimate	110871k	7900

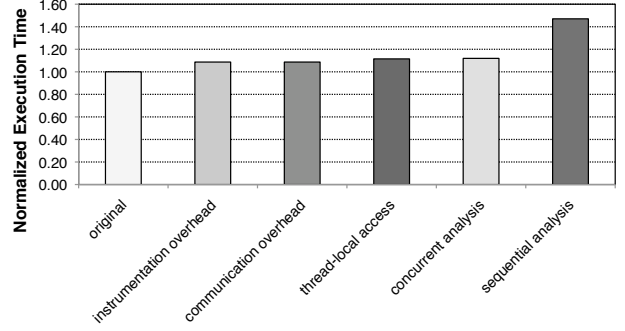
Table 1. Instrumentation frequency for each benchmark

The analysis monitor periodically monitors the pace of the application and analysis threads, detects performance bottlenecks, and remaps the tiles as shown in Figure 3. If the analysis is bounded by communication, the analysis threads need to spread out across cores to reduce the communication latency. If communication happens more frequently between the application threads or between the analysis threads, grouping each type of thread closely will improve locality and performance.

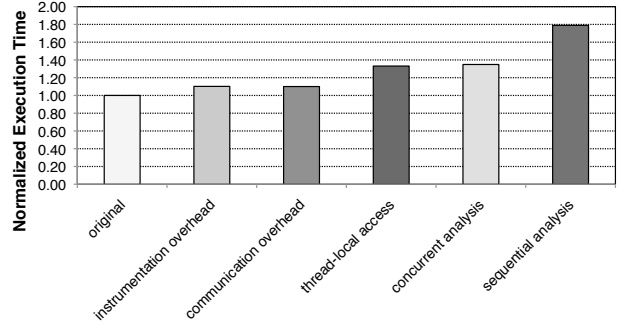
Our model is generally applicable to any dynamic analysis system, but there are two benefits to running it on a processor with explicit inter-core communication. First, fine-grained sharing between the application and the analysis thread does not pollute the cache. Therefore, overhead can be lower and more predictable. Second, since there is no buffering involved, total memory space usage for the analysis is scalable. If the producer and consumer are implemented using thread local buffers, total buffer memory will be proportional to the number of threads.

4. Case Study: Call Graph Profiling

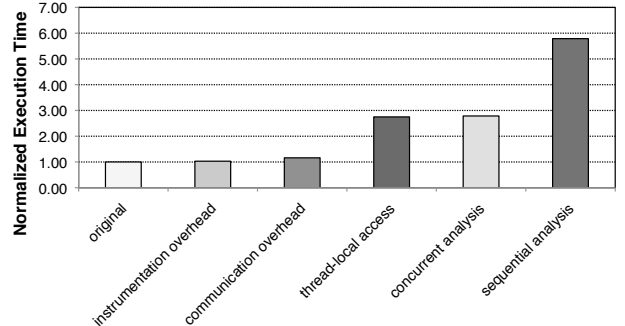
To evaluate the performance characteristics of explicit inter-core communication, we implement call graph profiling as a reference analysis. We modify *tile-cc*, a cross compiler for the TILE64 architecture that conforms to standard GCC, to instrument a call to each callback function on every function entry and exit. The compiler also passes the function id as an argument to each callback. Each callback sends the thread id and function id to the call graph profiler thread over the user dynamic network. Upon receiving data from a series of function calls and return traces, the call graph profiler keeps the calling context in memory and stores caller and callee information into a hash-table. Implementing a light-weight call graph is not our goal. Rather, our goal is to understand the performance characteristics when using the user dynamic network for dynamic analyses.



(a) bodytrack



(b) blackscholes



(c) fluidanimate

Figure 4. Call graph overhead breakdown

We compile the *bodytrack*, *blackscholes* and *fluidanimate* benchmarks from the Parsec benchmark suite [5] to the TILE64 architecture, instrumented method entry and exit with our modified compiler, and manually added callbacks on thread creation and termination. We used the *sim-medium* input that is included in the benchmark release and used 32 cores for the application and 1 core for call graph profiling. We left other cores for system-level threads like shepherds and a watchdog to reduce the perturbation.

Figure 4 presents the overhead breakdown of concurrent call graph profiling and also compares with running the same analysis code sequentially. Bar 1 shows the execution time of the unmodified code, which is our baseline. Other measurements are normalized to the baseline.

Bar 2 measures execution time when the callback functions are instrumented, but data is not transferred to the network. It presents the instrumentation overhead caused by calling the callback func-

tions, which is 3–9%. Our current implementation does not inline the callback function, so this overhead could be further reduced by implementing inlining.

Bar 3 shows the communication overhead on top of the instrumentation overhead (Bar 2), where the function id is sent to the call graph profiler on each callback, but the call graph profiler discards the data immediately. We do not send the thread id at Bar 3 to factor out thread-local access overhead. Therefore, (Bar 3 - Bar 2) represents pure communication overhead. It is remarkable that the communication overhead is less than 1% for `bodytrack` and `blackscholes`, and 13% for `fluidanimate`. The high overhead in `fluidanimate` is due to function call frequency. As shown in Table 1, `bodytrack` and `blackscholes` have similar function call frequencies, but `fluidanimate` has around six times the function calls per cycle.

Bar 4 shows the overhead of sending the thread id with the function id. Since the thread id and function id are both 16 bits, they are packed into a single word, so that the total data size transferred remains the same as in Bar 3. Most of the overhead comes from accessing the thread id from thread-local storage. The only differences between Bar 2 and Bar 3 are from the thread-local load operations and packing the thread id and function id values. The current thread-local storage implementation is inefficient, having unnecessary memory operations and function calls, which result in high overhead. We plan to improve this overhead by reserving a single general-purpose register for the base address for thread-local storage, so thread-local variables can be accessed with an offset from the base register.

Bar 5 shows normal concurrent call graph profiling overhead, and Bar 6 presents the overhead of running the same profiling sequentially, i.e. where call graph computation happens inside the callback function. The difference between Bar 2 and Bar 6 shows the computational overhead of generating the call graph (38%, 69%, 470%). Note that this overhead is almost hidden by running it on a separate core (only a 1% increase from Bar 4 to Bar 5). Because of the frequency of function calls, `fluidanimate` suffers from cache pollution in the sequential call graph implementation. Hash-table updates cause more cache misses, perturbing application performance. The computational overhead is 4.7 times greater than the 2x of the baseline, but the concurrent call graph kept pace with the benchmark because the hash-table update overhead is reduced by improved locality.

Overall, these results are promising and show that the interconnect network is useful for offloading analysis data from the application and introducing little additional overhead. While we have not performed a direct comparison, we have shown that the communication overhead introduced by our method, which directly uses inter-core interconnects, is less than that incurred when traditional cache coherency is used. These techniques show promise for enabling scalable concurrent and parallel dynamic analysis techniques through the interconnect network.

5. Conclusion and Future Work

We have shown the benefit and efficacy of using the explicit inter-core communication for dynamic analysis. When using the interconnect network, communication overhead for offloading the data from the application is minor, and does not pollute the cache. In the future, we will further investigate how to efficiently design scalable dynamic analysis techniques and will implement a practical system on top of our framework.

Acknowledgments

We would like to thank the Tilera Corporation for helpful technical discussions and for reviewing and providing feedback on this paper.

References

- [1] M. Arnold and D. Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *International Symposium on Code Generation and Optimization*, pages 51–62, San Jose, CA, Mar. 2005.
- [2] M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *ACM Conference on Programming Language Design and Implementation*, pages 168–179, Snowbird, UT, June 2001.
- [3] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, Oct. 2000.
- [4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, EECS Department, December 2006.
- [5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.
- [6] M. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *ACM Conference on Programming Language Design and Implementation*, Toronto, Canada, June 2010.
- [7] M. D. Bond and K. S. McKinley. Continuous Path and Edge Profiling. In *ACM/IEEE International Symposium on Microarchitecture*, pages 130–140, Barcelona, Spain, Nov. 2005.
- [8] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved Error Reporting for Software that uses Black-Box Components. In *ACM Conference on Programming Language Design and Implementation*, pages 101–111, San Diego, CA, 2007.
- [9] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A Concurrent Dynamic Analysis Framework for Multicore Hardware. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 155–174, Orlando, FL, October 2009.
- [10] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 117–126, December 2001.
- [11] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, San Jose, CA, 2006.
- [12] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical Fine-grained Decentralized Information Flow Control. In *ACM Conference on Programming Language Design and Implementation*, pages 63–74, Dublin, Ireland, 2009.
- [13] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. HeapMon: a Helper-thread Approach to Programmable, Automatic, and Low-overhead Memory Bug Detection. *IBM Journal of Research and Development*, 50(2/3):261–275, 2006.
- [14] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *ACM/IEEE International Symposium on Computer Architecture*, pages 224–235, München, Germany, June 2004.