Platform-independent analysis of function-level communication in workloads

Siddharth Nilakantan, Mark Hempstead
Department of Electrical and Computer Engineering
Drexel University
Philadelphia, PA USA
Email: {sn446,mdh77}@drexel.edu

***Abstract*—The emergence of many-core and heterogeneous multicore processors has meant that data communication patterns increasingly determine application performance. Microprocessor designers need tools that can extract and represent these producer-consumer relationships for a workload to aid them in a wide range of tasks including hardware-software co-design, software partitioning, and application performance optimization. This paper presents Sigil, a profiling tool that can extract communication patterns within a workload independent of hardware characteristics. We show how our methodology can extract the *true* costs of communication within a workload by distinguishing between unique, local, and total communication. We describe the implementation and performance of Sigil as well as the results of several case studies.**

***Keywords*-data flow graph; data dependencies; re-use; partitioning; critical path; function**

I. Introduction

The performance of microprocessors is limited by communication. This limitation, sometimes referred to as the *memory wall*, often refers to the cost of communicating with memory (hardware-level communication). Recent studies have found that the promise of speedup from technology scaling [1] or heterogeneous processors, such as GPUs, is diminished when hardware communication costs are included [2]–[4]. Continuing exponential performance scaling trends requires studying the effect of communication on hardware design.

Communication at a hardware-level is a run-time manifestation of communication at a software-level. Software-level communication refers to messages between software entities such as functions, threads, basic blocks, or even instructions. Due to its platform-independent nature, software-level communication can be useful in a variety of ways. A range of hardware and software tasks—including software development, parallel programming, hardware-software partitioning, and the design of network-on-chip—can be improved with a detailed understanding of software-level communication within a workload [5]–[7]. This work addresses the challenge of characterizing the sources and patterns of software-level communication in a workload; it does so in an automated way with low overhead.

The methodology presented in this work extracts architecture-agnostic properties of the workload such as control data flow graphs, dynamic dependency chains and data re-use lifetimes. These properties can help the hardware design process at an early stage, while also providing useful insights for software optimization. While methodologies and models exist that characterize memory access patterns, many of the profiles recorded by these methodologies are platform-dependent [8]–[11]. For example, the measured communication might depend on cache-size, cache configurations or other details of the platform's memory hierarchy and interconnection network.

Like most software profilers (e.g. *gprof*), we aggregate costs on a per-function basis, because functions define clear logical boundaries that are understandable to the software developer. When a profiler analyzes function-level communication, not all of the total bytes read and written by a function should be treated equally. Our methodology tracks the data produced and consumed by each function call and differentiates the first-time use from the re-use of bytes. We also distinguish between communication external to the function and local communication within the function by tracking the producer and consumer of each unique data byte in the program. Using the technique of *shadow memory* [12] we are able to index our table of functions efficiently without exploding state.

This paper also presents a custom tool we named "Sigil" that implements the shadow memory technique. Sigil leverages Dynamic Binary Instrumentation (DBI) technology and is implemented on top of Valgrind's Callgrind framework [13], [14]. Sigil can represent its profiling results in one of two ways: it can dump aggregates on a per-function basis or list the execution as a sequence of dependent "events." The latter representation allows a system designer to view a workload as a list of function calls connected by data transfer edges. Viewing the results using the "event" representation is more conductive to solving problems such as scheduling using critical path analysis [15], [16].

We demonstrate the utility of the tool by studying the characteristics of serial versions of workloads in the PARSEC benchmark suite [17]. We also show how Sigil can be used to drill down into a workload and discover the source of performance-limiting communication patterns.

Contributions:

1) Unique profiling methodology to automatically infer data dependencies of a program at the function-level.

978-1-4799-055-3/13/$31.00 ©2013 IEEE

2) A lightweight and minimal overhead implementation built on the Valgrind framework.
3) A method for interpreting and post-processing aggregate and event file data.
4) Case studies:
   a) HW/SW partitioning using trimmed calltrees.
   b) Workload data-reuse characterization.
   c) Detection of dynamic data-dependency chains used to infer parallelism and critical paths.

II. Sigil: Modeling data exchange in workloads

A. Collected Data

Sigil captures communication by tracking the producer and all consumers of every data byte generated by a program. Any self contained fragment of code can be a producer or consumer; basic blocks, functions, threads and even individual instructions can all be uniquely identified as data producing and consuming entities. In this work, we study communication between functions, as they provide a clear interface with software.

Designers will find that for some tasks, such as hardware partitioning, the *classification* of communication into categories is more useful than just recording the aggregates. Sigil classifies every communicated byte into two different categories: 1) *input/output/local* and 2) *unique/non-unique*. In the first category, *local* indicates that the byte was generated and read by the same function. The *Input/Output* identifier indicates that the byte was generated by one function and read by another. The *unique/non-unique* category of classification is used to distinguish between the first time use of a byte and subsequent re-use of it. *Unique* indicates that the consumer is reading this byte for the first time, while non-unique indicates that the consumer has read this same byte before.

Prior work has analyzed communication between functions [18], but does not distinguish *total* communication from *unique* communication. In their work, first time accesses to a byte of data are aggregated along with subsequent accesses to the same byte, not allow us to isolate the true read and write set of a function. In contrast, the *unique* byte counts from Sigil's profile determine the true inputs needed by a function.

The distinction between unique and non-unique communication is particularly important for HW/SW partitioning. A well designed accelerator (ASIC, GPU, or FPGA) for a function will include an internal buffer and will not repeatedly fetch the same data from memory. *Unique* communication is the true amount of data an accelerator needs to complete its task. Sigil not only captures data communicated between functions, but also local data generated within the function itself. In a HW/SW partitioning context, local data bytes will either be consumed within the pipeline of the accelerator or stored in local memory depending on the data re-use characteristics and the accelerator pipeline implementation.

Table I: Shadow Object Contents

Baseline		
variable	**size**	**description**
last_writer	8B	pointer to function
last_reader	8B	pointer to function
last_reader_call	8B	call number
Additional variables for Reuse mode		
re-use_count	8B	# of times byte was accessed
re-use_lifetime_start	8B	first access timestamp
re-use_lifetime_finish	8B	final access timestamp

We can study non-unique communication in a function to understand it's data re-use pattern. To facilitate this, Sigil also records statistics for each data byte: the number of non-unique accesses (re-use count) and the time between first and last non-unique accesses (reuse lifetime). This can give us hints into a function's impact on a memory system, showing how often a function is accessing the same data and the liveliness of that data.

Finally, Sigil captures the execution-sequence of functions in the program, including through calls and returns. For simplicity of analysis, we do not distinguish the order of events *within* a function but do capture the order of events between functions. While it is inside a particular function, Sigil dumps the communication events that occur. This is useful in building dependency chains to help determine function level parallelism and critical path lengths of a program.

B. Measurement Methodology

Sigil uses a shadow memory implementation to keep track of the producers and consumers of every data byte in the program. The goal of memory shadowing is to hold a shadow data object for every unique byte used by the program. Shadow objects are not visible to the binary being profiled and do not affect the correctness of the program. Sigil's shadow memory structure is derived from Nethercote and Seward's description [12]. It is a two-level table, similar to an operating system page-table, where each level is indexed by a portion of the data byte-address. The second-level structures are created only when the corresponding portions of the address space are accessed. These second-level structures are a chunk of shadow objects which are initialized to "invalid" until the data byte corresponding to those addresses are used by the binary.

The content of a shadow object is shown in Table I. The *baseline* variables collected for all workloads allow Sigil to determine producer/consumer relationships. When Sigil operates in re-use mode, the shadow memory object is extended with additional variables used to derive data liveliness and re-use. When a write occurs to a particular address, Sigil looks up the corresponding shadow object and marks the function doing the write as the last writer. When a read to the same address occurs, Sigil locates the shadow memory element and infers the source (i.e. last

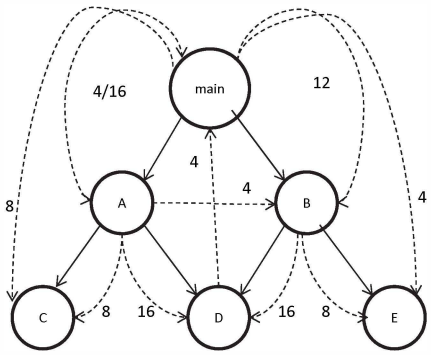

Figure 1: Data and control flow between functions

writer function ID) of the data byte. In addition, Sigil uses a pointer to the *last reader* to distinguish between unique reads and non-unique reads. If a subsequent read occurs to a data element, Sigil checks if the reading function is the last reader and if so counts the read as non-unique.

Sigil can represent output data in two ways: (1) by reporting the aggregates of measured communication for each function in the program; (2) by recording a list of all of the data transfers that occur. In the latter representation, a program's essence can be reconstructed as a sequence of dependent "events". These events are fragments of computation separated by data transfer edges. Note: the producing and consuming entities are still functions, and this representation helps designers understand the order imposed on function calls due to the algorithm implemented by the program.

C. Interpreting Sigil Results

1) Processing calltrees with dependencies: Figure 1 shows a sample control data flow graph for a *toy program* generated using Sigil's profiling data. This graph is essentially a calltree with edges representing dependencies and the graph nodes represent functions. Henceforth, the term "control data flow graph" refers to a calltree with dependencies. Call edges are represented by the bold edges and data dependencies are represented by the dashed edges. The directed data dependency edges are weighted by the number of bytes needed by the receiving function.

Using control data flow graphs for partitioning:

Task graphs have proved useful in a variety of ways, including schedule optimization and HW/SW partitioning [6], [11], [19], [20]. The goal of partitioning is to select a subset of tasks to be offloaded to ASICs or FPGAs. The tree is sliced into collections of nodes, such that communication between the different collections is minimal. Task graphs usually represent a sequence of dependent tasks. The tasks themselves are self-contained and must execute completely before a dependent task can begin. The notion of a task is best represented by a function in a software implementation as functions are frequently re-used tasks [21]. However, functions are not self-contained as they make calls to other functions before returning. Thus, when exploring partitioning problems, we cannot simply view functions as abstract tasks.

Given a control data flow graph (calltree with dependencies), determining the granularity of merging nodes is one of the key questions when slicing the tree; if we only include the logic within the function itself, then calls to the subtree would incur the cost of communication from either a general purpose core or the cost of sending data to separate accelerators. Thus, an accelerator designed for a function node in the calltree should include *all* of the functions in the sub-tree to absorb the cost of communication. This model assumes that an accelerator is non-preemptible and that all input data must be ready before it begins execution for a call.

We illustrate the process of merging nodes in Figure 2. Figure 2 shows the control data flow graph of the same *toy program* used in Figure 1. We determine costs at different granularities by drawing a box around the functions to represent hardware functionality for the entire set of functions. Based on what we mentioned above, we draw boxes around a node and its entire sub-tree. Any dashed edges within the box are then discarded and edges flowing in/out of the box are accumulated into the communication cost of the parent node. We sum measurements such as computing operations and CPU memory traffic to provide the software and platform-independent costs for the node. We call the accumulated costs for a node the *inclusive cost* of communication and computation for the entire sub-tree. For simplicity, Figure 2 does not show the computation and local communication costs of each function, but these metrics are also captured by Sigil.

Figure 2a shows the calltree before merging nodes. Note: we have separated costs for function D based on the context it is called from, and attributed each context to the distinct nodes D1 and D2. If node A is selected for merging, then we draw a box that encompasses its entire sub-tree as shown in Figure 2b. We attribute only communication outside the box to A. We represent the computation of the merged subtree by summing the total number of operations for the entire sub-tree, resulting in a trimmed control data flow graph with five nodes.

Metric for partitioning Given a control data flow graph, we must trim the calltree by merging nodes such that the leaf nodes of the resulting tree are accelerator candidates. For the purpose of demonstration, we developed a simple algorithm to traverse a calltree and merge functions according to a heuristic. We characterize every function in the calltree with several parameters: 1) an estimated software run time calculated by Callgrind, 2) the number of operations in the function, and 3) the hardware offload time, calculated as the time to communicate data to and from the accelerator assuming a fixed SoC bus bandwidth. These parameters are derived from the *inclusive costs* and are used by the

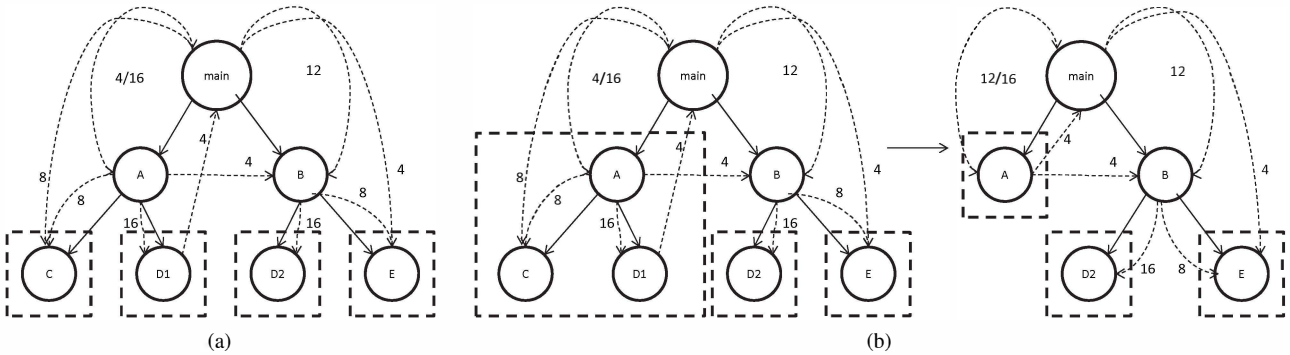

Figure 2: Partitioning control data flow graphs where nodes represent functions

heuristic.

We define a metric *breakeven-speedup* to determine if a node sub-tree should be merged. Breakeven-speedup, shown in equation 1, is *the computational speedup that an accelerator for a particular function would require* in order to offset the data-offload costs for input, $t_{comm:ip:accel}$, and output, $t_{comm:op:accel}$. Any computational speedup obtained in excess of the breakeven-speedup will result in an overall improvement in execution time. Determining if the breakeven-speedup for a function can be achieved depends on the amenability of the function logic to a hardware implementation. We leave the investigation of mapping candidate functions to specific hardware implementations to future work.

$$S_{breakeven} = \frac{t_{sw}}{t_{sw} - (t_{comm:ip:accel} + t_{comm:op:accel})} \quad (1)$$

The goal of the heuristic is to minimize the breakeven-speedup of all the leaf nodes of a trimmed calltree. Each branch of the trimmed calltree should have the least breakeven-speedup at the bottom of the branch. The heuristic is thus optimized for maximum application coverage with useful functions—$i.e.$ Amdahl's law: the ratio of execution time in the candidate function over the total execution time of the workload— and for minimal communication.

2) Processing Sequential Event Files with Dependencies: The second form of output we can process is an event-file which maintains the sequence of operations in a program. We can post-process these files to separate the dependent chains of events in the program. These dependent chains reveal the critical path of an application and the theoretical limits of scheduling parallel tasks.

In this subsection, we show a simple example of how this can be accomplished. Figure 3 illustrates how we construct dependency chains of events for the same *toy program* discussed in the previous subsection. As nodes get updated or added to each chain, we must re-calculate the critical path. Each node in the figure represents a single function call. The self-cost of each node, shown inside the box, is the number of operations performed within the call. The inclusive cost, shown outside the box of a node, represents the sum of the self-costs of the longest chain from "main" to that node. The longest chain in the entire tree is the critical path. The critical path is highlighted with nodes in gray and edges in bold. In the example, A and C are encountered first with A preceding C. Both are attached to main and the path through C is the critical path. Looking at the calltree in Figure 1, A calls C and when C returns, we encounter A again. We model functions as non-blocking, so that they can potentially run in parallel and start consuming data. To include the effect of this, we add the second occurrence of A as a separate node although it belongs to the same call, so as to not affect the inclusive cost of C. We also add a dependency link to the previous occurrence of A to conservatively enforce order between regions within A. Node D is then added when it consumes data from that particular call of A. The path to C through A is the updated critical path. Finally, when a link is established between C and D, the critical path is now updated to include D as the leaf node.

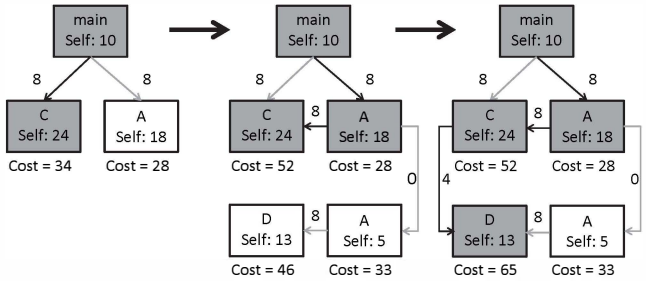

Figure 3: Communication between parallel paths

III. Sigil Implementation

Our profiling tool, Sigil, was written on top of Callgrind (part of the Valgrind instrumentation framework) [13]. Sigil captures the data transfer that occurs between any of the functions in the user program, including data passed by reference. We also capture the number of calls to each function to determine the average cost of a single call. We keep separate accounting of costs for functions called through different contexts. These measured characteristics

are application-specific and independent of the platform and processor architecture.

Valgrind is a DBI framework that is capable of intercepting a user program at run time and provides mechanisms to perform heavyweight analysis of the program [13]. Valgrind translates assembly into an intermediate representation. This representation reduces the program to a collection of primitives such as memory accesses and operations.

Callgrind is a tool that is built over the Valgrind framework [14]. Callgrind captures a calltree of the running programs and also performs on-the-fly cache simulations to determine the behavior of the program. It maintains costs for each function in the calltree of the running program. A programmer can identify performance bottlenecks in a software application by using a breakdown from Callgrind, of parameters such as cache misses and branch mispredictions. We use some of the metrics captured by Callgrind in the Case Study section of this work to estimate the execution time of a function run on a general purpose CPU. These default Callgrind profiling parameters include miss rate, branch misprediction rate, and instruction count. Our performance estimation formula matches the calculation used by Callgrind to estimate cycle count.

Sigil hooks into Callgrind to identify function names, obtain addresses and count operations. In general Sigil can use any framework that identifies communicating entities, and exposes addresses and operations to the tool. Callgrind was minimally modified to insert calls to Sigil and allow it to compile along with Callgrind. The biggest change made includes the functionality to log floating point and integer operations within Callgrind. As with any Valgrind tool, Sigil's efficacy is drastically reduced when the binary does not have debugging symbols. The binary, otherwise, can remain unmodified.

System calls, because they are not completely visible to Valgrind, must have special handling. Sigil is able to capture the names of system calls and capture the input and output bytes but not see the detailed memory and communication used inside the system call.

A. Sigil Characterization

Sigil incurs a larger slowdown than Callgrind over native runs of benchmarks. Sigil uses more memory and incurs more memory lookups than Callgrind as it shadows the entire program state. We believe this overhead is justified as Sigil captures platform-independent data and only needs to be run once.

With data-re-use monitoring enabled, Sigil's memory usage is up to 2 times larger when the instrumented program touches a large range of addresses. We have added a simple FIFO mechanism to free up space from shadow bytes of addresses that have been least recently touched by the program. Using this option improves performance when instrumenting programs with large memory usage. This memory limit parameter is a command line option. *dedup* is the only benchmark amongst the PARSEC benchmarks, for which we have needed to enable this memory limit parameter. We found the corresponding loss of accuracy to be negligible.

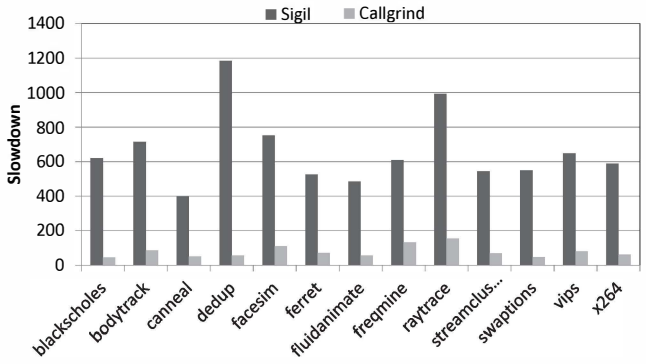

Figure 4: Slowdown of Sigil and Callgrind relative to native for baseline function-level profiling

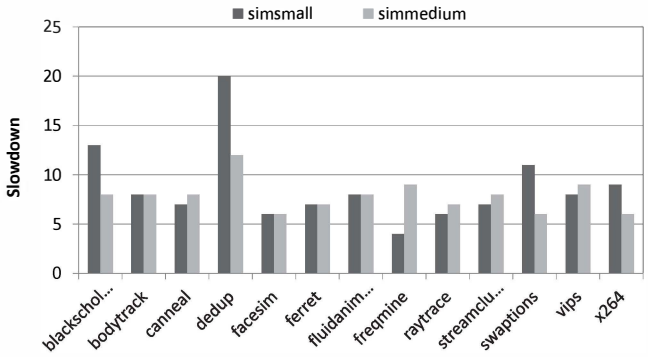

Figure 5: Slowdown of Sigil relative to Callgrind for baseline function-level profiling

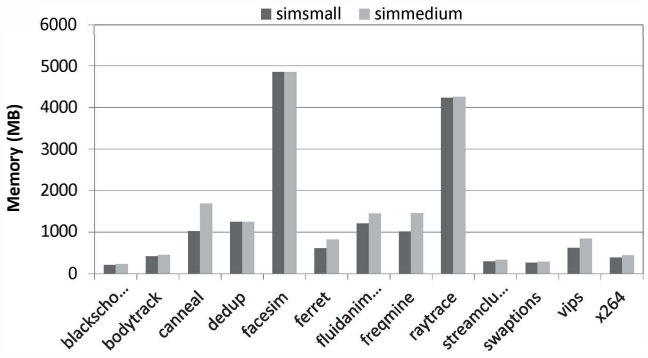

Figure 6: Memory usage for baseline function-level profiling

We measured the cost of running Sigil on an Intel Xeon E5620 platform with 24GB of DRAM. Figure 4 shows the function-level profiling slowdown of Sigil and Callgrind relative to native runs without any instrumentation of the serial version of PARSEC workloads with the "simsmall" input. The slowdown is much larger compared to Callgrind; the average slowdown being 580x for simsmall inputs and 720x for simmedium inputs. Figure 5 shows the slowdown of

Sigil relative to Callgrind; we observe an average slowdown of 8-9x and remains fairly consistent given Sigil's ambitious goals. *dedup* is an outlier which incurred more slowdown as we enabled the memory limiting command line option to keep Sigil's memory usage manageable. *blackscholes* and *swaptions* with simsmall inputs take very little time in both frameworks (less than 5 minutes). Figure 6 shows the memory usage of Sigil for workloads as we increase the datasize. The memory increase also remains consistent for increased datasize. *facesim* and *raytrace* are intensive benchmarks that use larger amounts of memory but incur constant overhead over a native run.

IV. Usage Case studies

In this section, we show how Sigil's captured information can be used to gain insight into the data usage of workloads by performing the following case studies:

1) HW/SW partitioning of control data flow graphs (call-trees with dependencies)
2) Data re-use of serial versions of PARSEC benchmarks
3) Critical path analysis of serial versions of PARSEC benchmarks

A. Control Data flow graph partitioning

With the growing popularity of multicore processors—including, more recently heterogeneous processors—deciding how to partition the workload across multiple general-purpose cores and/or fixed-function accelerators is challenging. Partitioning is easy with Sigil because the information collected from a running binary is closely related to the source-code level implementation. The control data flow graphs constructed from Sigil's profile data for a program, represent the producer and consumer relationships between functions annotated with the amount of unique communication. As explained earlier, the data flow edges in the graph must be unique communication as an accelerator with internal memories would not incur costs for non-unique communication. We apply the post-processing technique described in Section II-C1 to perform communication aware partitioning and function selection. The selected functions are listed as potential candidates for acceleration.

We ran Sigil on a number of PARSEC benchmarks and used the heuristic-based granularity metric to trim the control data flow graph for each benchmark. The heuristic naturally tries to merge sub-trees to maximize coverage while minimizing communication to the merged node. In this example, we consider the leaf nodes in the trimmed calltree as "selected" as tentative candidates for hardware acceleration. Figure 7 shows the breakdown of an application's native execution time by fraction of candidate functions. The coverage represented by the leaf nodes of the trimmed calltree is the lower bar and the rest of the application is the upper bar.

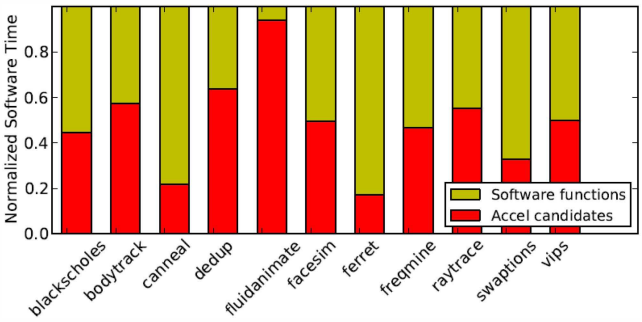

Figure 7: The normalized coverage of the leaf nodes of the calltree for all benchmarks

From the graph, we see that many applications spend over 50% of their execution in the leaf nodes of the trimmed calltree. The exceptions are Canneal, Ferret and Swaptions, whose candidate functions show low "coverage" of the overall application in terms of execution time. Functions with low coverage indicate fewer "hot code" regions.

For a designer to evaluate the best functions for acceleration first, we must sort the functions by their breakeven-speedup. Table II shows the top functions picked by our proposed max-coverage, min-communication heuristic from a few PARSEC-2.1 benchmarks. These functions are listed from the top to bottom in order of increasing breakeven-speedup. A low breakeven-speedup indicates a small communication cost to offload computation. We find that the breakeven-speedup in most cases for the top few functions are close to 1. Table III shows the breakeven-speedups for the bottom few functions. It can be seen that the functions are mostly utility functions such as constructors(e.g. std::vector), destructors (e.g. free) and initializers (e.g. std::string::assign). These same functions also exhibit less computational intensity. To illustrate the usefulness of functions picked by our heuristic, we describe a subset of them here:

1) *ieee754_(operation)*: These functions are part of the IEEE 'math' library. These are usually very fast code implementations with existing hardware support.
2) *mul/mpn_mul*: These are multiplication calls to the math library. While direct hardware support exists in contemporary processors, these calls are made for compatibility purposes.
3) *ImageMeasurements::ImageErrorInside*: In the bodytrack benchmark, a human body is tracked with multiple cameras through an image sequence. This function measures the "Silhouette" error of a complete body on all camera images.
4) *FlexImage::Set*: This bodytrack function initializes an image and is mostly composed of memcopy calls.
5) *memchr*: This is a library call which searches for a character in a block of memory.
6) *std::string::compare*: This call compares two strings.

Table II: Breakeven speedup for top 5 functions for PARSEC-2.1 benchmarks with simsmall input

Blackscholes	S(breakeven)	Bodytrack	S(breakeven)	Canneal	S(breakeven)	Dedup	S(breakeven)
_strtof	1.006	FlexImage::Set	1.000	_mul	1.008	sha1_block_data_order	1.008
_ieee754_exp	1.011	_ieee754_log	1.007	memchr	1.028	sha1_block_data_order	1.013
_ieee754_expf	1.019	_ieee754_log	1.007	netlist::swap_locations	1.040	_tr_flush_block	1.013
_ieee754_logf	1.021	IM::ImageErrorInside	1.007	memmove	1.057	write_file	1.033
_mpn_mul	1.039	IM::ImageErrorInside	1.007	std::string::compare	1.089	adler32	1.041

Table III: Breakeven speedup for worst 5 functions for PARSEC-2.1 benchmarks with simsmall input

Blackscholes	S(breakeven)	Bodytrack	S(breakeven)	Canneal	S(breakeven)	Dedup	S(breakeven)
_dl_addr	1.961	std::vector	1.278	_gnu_cxx	7.466	memcpy	6.119
_mpn_rshift	1.631	_IO_file_xsgetn	1.266	std::locale::locale	3.136	memcpy	1.811
_IO_sputbackc	1.421	DMatrix	1.143	std::string::assign	2.645	hashtable_search	1.441
free	1.238	DMatrix	1.143	std::basic_string	1.893	hashtable_search	1.433
_mpn_lshift	1.206	isnan	1.098	operator new	1.609	free	1.156

7) *adler32*: A checksum algorithm optimized for speed over accuracy.
8) *_tr_flush_block*: Part of the zlib algorithm implementing the flushing mechanism.
9) *sha1_block_data_order*: This call is the core of the SHA1 calculation.
10) *netlist::swap_locations*: This call swaps two vectors.

There are a few functions in the list that will benefit from accelerated communication rather than computation. *FlexImage::Set* from the bodytrack benchmark is one such example and it is composed of "memcpy" calls. Since breakeven-speedup focuses on minimizing communication, it flags *FlexImage::Set* as having very low communication as all the communication with memcpy is absorbed when calculating *inclusive* costs. For example, *FlexImage::Set* can potentially be sped up by using memcpy accelerators [22].

This study shows that, with preliminary knowledge of a target platform and a little workload analysis on a collection of workloads, we can determine a reasonable list of functions to target for acceleration. Prior work has used Sigil's data to select functions for acceleration and estimate performance [23]. Note: this methodology is more effective when the profiled code is more modular and does not deviate significantly in behavior between calls to the same function. The next natural step for a system designer would be to traverse the list, apply system constraints and perform an amenability test of these functions to determine if they can be accelerated on hardware and for what cost.

B. Data Reuse

We characterize a data byte by its re-use lifetime in the program and the number of times it is re-used. Researchers have shown that taking advantage of data re-use behavior can enhance the performance in a range of areas from FPGA implementations, memory systems, and loops in scientific applications [24], [25]. In this Section we study the data re-use patterns of PARSEC benchmarks in an architecture agnostic manner. Sigil provides an automated way of capturing and analyzing data re-use at the function-level with no prior knowledge of the application. We define *re-use lifetime*

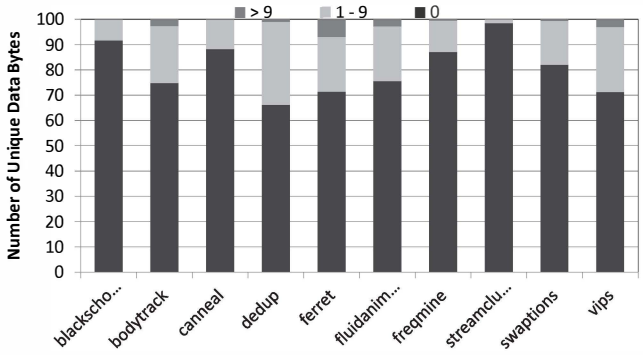

Figure 8: Breakdown of data bytes based on re-use counts for PARSEC benchmarks (simsmall input)

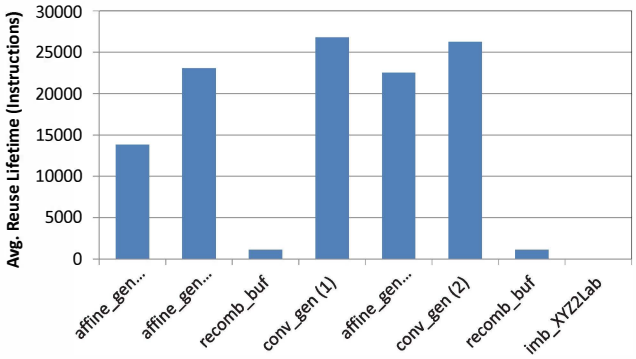

Figure 9: Average re-use lifetimes of the top *vips* functions by number of data bytes reused

as the time between the first and last read of a single data byte within a function call. In order to remain architecture independent, we use the number of retired instructions as a proxy for execution time.

1) Data Reuse Within a Benchmark: We use Sigil to study the data re-use of PARSEC benchmarks, first in aggregate and then zooming in to specific functions of interest. Figure 8 shows the breakdown of repeat accesses to data for several PARSEC benchmarks with simsmall inputs. The accesses are categorized based on the number of times each byte is re-used. The bottom-most section of

each bar indicates zero re-use (the object is written once and read only once within each function it is accessed in), while the remaining stacked bars represent two ranges of re-use: between 1 and 9 accesses, and greater than 9. We see that for most benchmarks a very small percentage of data elements are used more than 9 times. As a significant percentage of data is created and consumed without ever being read again, most intermediate data generated by these benchmarks are consumed quickly and need not be cached at all. Functions with limited re-use, such as those in the blackscholes and streamcluster benchmarks, take very little advantage of the cache in general. However, if the accessed data is not too sparse, such functions can still benefit from the spatial locality extracting properties of a cache-based hierarchy, such as large cache line sizes and prefetching. We hypothesize that applications with limited re-use could benefit from custom memory systems, incorporating temporary buffers with explicit eviction of data when it is dead. We increased the workload dataset size and found the simmedium and simlarge inputs of PARSEC have almost identical distributions. We have omitted these figures because of space constraints.

Re-use lifetime is an indicator of the time for which data needs to reside in memory during program execution. This analysis is important to SoC hardware designers who need to size buffers and scratch pad memories for accelerated functions. Using data from Sigil we can trace the source of re-use in a benchmark of interest, e.g. *vips*. We sort the functions in *vips* based on their contribution to the total amount of data re-use. Next, to understand the implication of large re-use, we look at the top list of functions and examine the average lifetime of a re-used data byte (reused at least once) in those functions. This is shown in figure 9. Since Sigil keeps separate accounting of functions called for different contexts, some functions occur more than once in the figure and are distinguished by the number in parentheses. Functions with large average data re-use lifetimes may not need to be cached as their data will be evicted before they are reused anyway.

In *vips*,the "conv_gen(1)" function has the highest and "imb_XYZ2Lab" has the smallest average re-use lifetime. These two functions and the "affine_gen" functions are the three biggest contributors to the total unique data bytes processed by the benchmark (the total includes the input data, and locally generated data), with each of their individual contributions being close to 10% each. The remaining unique data bytes are distributed across numerous functions with most of their contributions being close to 2 - 3 %. Since "conv_gen" and "imb_XYZ2Lab" are such large contributors to the overall data and incur such varying re-use lifetimes, we investigate them further.

2) Data Reuse Within A Function: Sigil can also capture a histogram of data-re-use during a function call. Each bin in the histogram corresponds to a range of re-use lifetimes and the value of that bin is the count of data bytes whose re-use lifetimes fell in that range. This information can help designers understand cache behavior and potentially design custom memory systems. Figures 10 and 11 shows the histogram for the "conv_gen" and "imb_XYZ2Lab" functions in *vips* respectively, with the y-axis in logarithmic scale. In "conv_gen", the distribution has a long tail and a central peak while "imb_XYZ2Lab" has a peak at 0 re-use and a short tail. The peak in "conv_gen" signifies that there are plenty of data elements that have large re-use lifetimes and hence bad temporal locality. For such functions, the cache size will heavily determine the performance of the function, and indeed, of the program.

Designers can explore dynamic methods of partitioning the cache into a scratch area and cache area to help such functions with large re-use lifetimes. In this case, a clever memory system would keep the data for this function in a scratchpad so as to not evict it until the function returns. Alternatively, designers can partition the cache into regions with different eviction rates i.e lazy eviction vs. fast eviction. A compiler hint or a runtime monitor could easily embed this information to ease memory partitioning decisions at run time. The "imb_XYZ2Lab" function reuses data at a higher frequency, which indicates increased temporal locality.

While we used a memory system with a cache as an example of gaining insight into memory behavior, the platform-independent nature of our data allows us to investigate the behavior of any arbitrary memory system. For instance, the data above is equally applicable in scenarios such as HW/SW Codesign and accelerator design. The re-use data captured by Sigil shows how many data bytes need to stay in an accelerator's local buffer after being consumed once. This will help determine buffer sizes based on an execution schedule for the function. For example, Cong et. al use the concept of BB-curves that indicate tradeoffs in increasing local buffer area for an accelerated function against external bandwidth pressure [26]. Such curves are a function of numerous variables besides data re-use, including the amount of parallelism available in the program and exploited in the accelerator implementation, the pipeline depth, and the initiation interval of the accelerator.

3) Data Reuse at Cache-line granularity: Byte-level re-use analysis is useful in understanding memory behavior on arbitrary memory systems, but needs to be used with a detailed model of execution and a hardware description. Sigil can also capture line-level re-use when configured with the cache line size. In this mode, Sigil shadows every *line* in memory rather than every *byte*. Our byte-level re-use characterization shadows every unique byte and accumulates costs at function-level granularity. In this mode we print re-use counts and lifetime for every block touched by the program, instead of aggregating costs by function.

The re-use behavior of cache lines is less architecture-independent but it can show a software developer or system

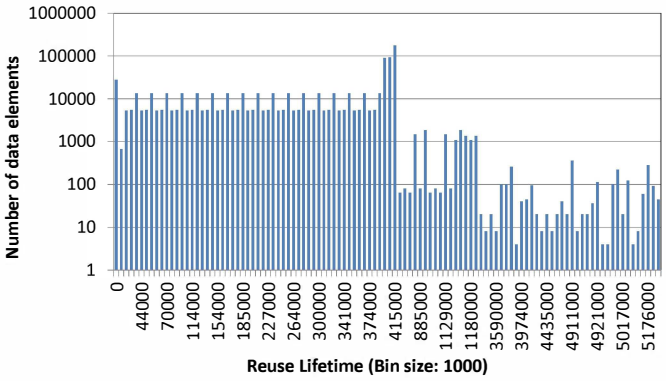

Figure 10: Data re-use distribution of "conv_gen" in *vips*

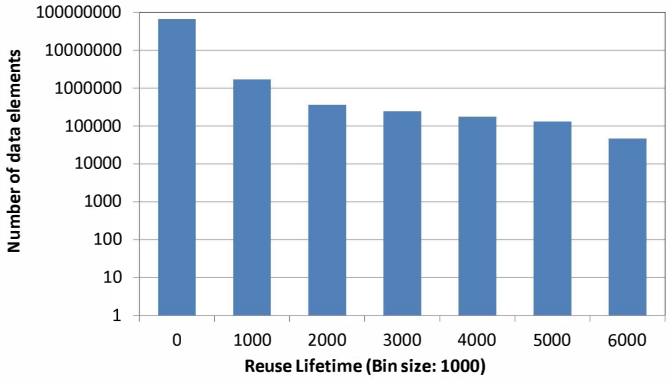

Figure 11: Data re-use distribution of "imb_XYZ2lab"in *vips*

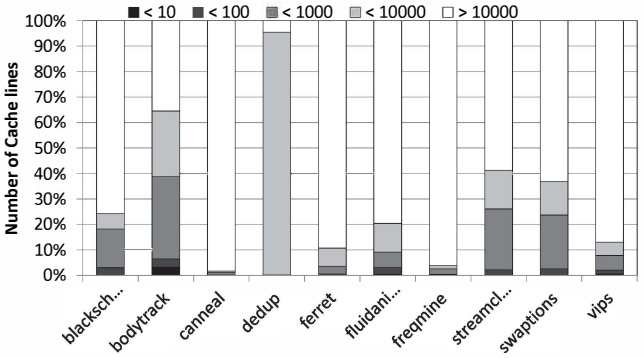

Figure 12: Breakdown of lines in memory based on re-use counts for benchmarks in the PARSEC Benchmark Suite (simsmall input)

designer how to optimize cache use or improve cache design. Figure 12 shows the breakdown of lines in memory by reuse count. While almost all benchmarks have lines re-used more than 10,000 times, Dedup, Bodytrack and Streamcluster have a significant number of lines that are re-used fewer times. Lines with low re-use counts across different data sizes can be marked as dead after they are fully reused. This information can be used for re-use distance analysis and to inform cache-replacement policies. There has been prior work exploring these techniques [25], [27] in detail, using information from the compiler or profiles collected from architectural simulation.

C. Critical Path Analysis

Critical path analysis has been applied to a range of domains from ASIC design, to scheduling, distributed systems, and networked systems [15], [16]. By measuring this path, the *critical path*, programmers and system designers can focus their design efforts on reducing the critical path and thus improving the functional parallelism of the workload. As explained in Section II-C2, the dependency tracking features of Sigil allow it to examine the dependencies between functions and discover the longest path of dependent functions within a program.

Using the information collected by Sigil, we construct dependency chains from the beginning of the program, following the methodology described in Section II-C2. The longest of these chains is the critical path. These paths could also represent the ideal execution schedule of computation events. As explained earlier, we distinguish between individual calls to a function by creating new nodes in the chain for every individual call. We also assume calls to child functions can be non-blocking and are only limited by their data dependencies. The maximum theoretical function-level parallelism is the ratio of overall serial length of the program to the critical path length. This ratio represents the limit to the extractable function-level parallelism in the program. We analyze the serial versions of a few PARSEC benchmarks and the libquantum benchmark from SPEC to establish their limit. The results are plotted in Figure 13.

To investigate further, we examine the functions in the critical path for streamcluster and fluidanimate benchmarks. We found the following functions in the critical path for streamcluster from (leaf to main):

drand48_iterate → *nrand48_r* → *lrand48* →
pkmedian → *localSearch* → *streamCluster* → *main*

Streamcluster is characterized by many short paths, where functions closer to the leaf-end of the critical path are of small consequence, e.g. rand. While the theoretical parallel limit is high due to the shortness of the individual paths, the overhead may not allow a programmer to extract all the function-level parallelism. We find a similar situation for libquantum as well.

The functions for fluidanimate are as follows:

ComputeForces → *main*

Fluidanimate's path is composed of a single function, ComputeForces. This function does the bulk of the work in fluidanimate, contributing close to 90% of the operations in the entire workload. As a result, a designer can speedup a program by accelerating/optimizing such a function with a goal of matching the other path lengths.

For the sake of simplicity, we do not employ more

sophisticated critical path analysis based on literature, which also take communication edges into account [16]. Besides highlighting the theoretical parallelism, we can use critical path information to build an optimal schedule for the program. The functions in parallel paths in a program can be mapped onto multiple cores such that dependencies are respected. A software developer may have a fixed number of scheduling slots based on the number of available cores. The developer can map dependency chains onto these slots so as to minimize communication between slots and balance the load among them.

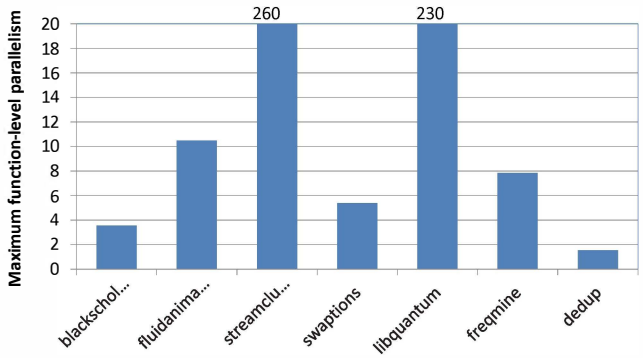

Figure 13: Maximum speedup based on function-level parallelism

V. Related Work

While methodologies and models exist that characterize hardware and task-level communication patterns [8], [10], [11], many of these profiles are very specific and the bytes of data transfer measured are very dependent on the characteristics of the platform's memory hierarchy and runtime behavior. Curreri et al, in particular, propose an automated methodology for capturing communication between application processes, but this do not distinguish between the first use and re-use of data [10].

Prior work in the hardware-software co-design field specifically use instructions, data flow analysis, and communication in the design process [19], [20]. These methodologies do consider the impact of communication on performance, but they do not extract data flow patterns from existing binaries automatically, which makes it difficult to apply the methodology to all workloads. Gremzow et al. employ dynamic instrumentation to determine both data flow between functions and reconstruct source/high level information to assist high level synthesis [18]. Galanis et al. [6] derive data flow graphs using static analysis and dynamic profiling of a given workload. However, neither work classify communication and account for unique data transfers.

Work in the reconfigurable computing field also explores the hardware-software partitioning problem. Smith and Peterson [5] propose a model that includes communication costs to estimate speedup of FPGA-accelerated cores for multi-threaded applications. The RC Amenability Test (RAT) from Holland et al. [28] describe models to quickly estimate the performance of applications targeted at FPGAs, while Huang et al. [11] propose splitting task graphs such that overall communication in the system is kept to a minimum. Although these models and techniques capture the impact of communication, they assume prior knowledge of the application or existing data flow graphs. In this work, we discuss an automated way of extracting communication and applying it on arbitrary workloads, using similar models. Sigil's profile has been used along with an assumed execution model to measure overall gains with offloaded functions [23].

Recently, tools have been released that use dependence analysis to highlight parallelism in loops [7], [29]. There has also been work that uses compressed traces to do dependence analysis and extract parallelism [30]. These works use traces or access histories for their dependence analysis, with the sole purpose of extracting parallelism. Sigil uses memory shadowing which allows it to accurately see dependencies across the workload and also classify it as unique and non-unique. Kremlin [31] identifies potential parallel regions of a given serial workload using hierarchical critical path analysis. These abstract regions do not have to be at function boundaries. Kremlin also do not classify communication as unique and non-unique. Gupta et. al [21] propose models to parallelize statically-sequential programs written in a suitable data flow fashion. However, their parallel executable functions are identified by programmers.

VI. Conclusion and Future Work

In this work we have presented a DBI based methodology to characterize software-level communication in an architecture-agnostic manner. Built on top of Callgrind, our tool Sigil implements the methodology. Sigil uses a memory shadowing technique that holds the producer, consumer and re-use information for every byte of the program. This technique incurs reasonable overhead for capturing platform independent software-level communication. It does not require any manual intervention such as modification of the source code or prior knowledge of the application.

We highlight the utility of the tool in HW/SW partitioning, data re-use analysis and critical path analysis. Using the profiled data dependencies between functions, we construct control data flow graphs, data dependency chains and re-use lifetime histograms. With the help of a simple heuristic, we demonstrate how the control data flow graphs can be used to partition and select functions that show promising characteristics for hardware acceleration. We show how the data re-use histograms in Sigil's profiles can be used to discover functions exhibiting good and bad temporal locality. We also comment on how this data can be used to infer a function's behavior on arbitrary memory systems. Finally, we estimate the maximum theoretical function-level

parallelism in a workload by constructing parallel paths from dependency chains and identifying the critical path.

Collecting profiles using Sigil incurs a large slowdown when compared to pure Callgrind profiles. However, these profiles are platform-independent and we only need to collect them once. As a result, the profiles will remain the same despite the platform that the profile is run on.

We will shortly release both the tool and post processing scripts to the wider research community. In addition, we plan to release the profile data for many commonly used benchmarks. As these profiles are platform independent, researchers can use the data without running Sigil.

ACKNOWLEDGMENT

The authors would like to thank the peers and reviewers for their comments and valuable feedback. The authors would also like to sincerely thank Steven Battle for his help with arranging content and the editing process of the paper. Help with diagrams and proofreading by the rest of the DPAC Lab (Jason Palazsewski, Rizwana Begum, Tianyun Zhang) is also appreciated. Finally, the authors would like to thank Tipp Moseley for shepherding the paper to completion.

REFERENCES

[1] G. Venkatesh, J. Sampson *et al.*, "Conservation cores: reducing the energy of mature computations," in *ASPLOS*, 2010.

[2] E. Chung, P. Milder, J. Hoe, and K. Mai, "Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?" in *MICRO*, 2010.

[3] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *ISPASS*, 2011.

[4] S. Nilakantan, S. Annangi, N. Gulati, K. Sangaiah, and M. Hempstead, "Evaluation of an accelerator architecture for speckle reducing anisotropic diffusion," in *CASES*, 2011.

[5] M. C. Smith and G. D. Peterson, "Parallel application performance on shared high performance reconfigurable computing resources," *Perform. Eval.*, May 2005.

[6] M. Galanis, G. Dimitroulakos, and C. Goutis, "Speedups from partitioning critical software parts to coarse-grain reconfigurable hardware," in *ASAP*, 2005.

[7] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *CGO*, 2009.

[8] Y. Kim and A. Shrivastava, "Cumapz: a tool to analyze memory access patterns in cuda," in *DAC*, 2011.

[9] S. Williams *et al.*, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, 2009.

[10] J. Curreri, G. Stitt, and A. George, "Communication visualization for bottleneck detection of high-level synthesis applications," in *FPGA*, 2012.

[11] M. Huang, V. K. Narayana *et al.*, "Reconfiguration and communication-aware task scheduling for high-performance reconfigurable computing," *TRETS*, 2010.

[12] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *VEE*, 2007.

[13] ——, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[14] J. Weidendorfer *et al.*, "A tool suite for simulation based analysis of memory access behavior," in *ICCS*, 2004.

[15] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge, "End-to-end performance forecasting: finding bottlenecks before they happen," in *ISCA*, 2009.

[16] A. Saidi, N. Binkert, S. Reinhardt, and T. Mudge, "Full-system critical path analysis," in *ISPASS*, 2008.

[17] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[18] C. Gremzow, "Compiled low-level virtual instruction set simulation and profiling for code partitioning and asip-synthesis in hardware/software co-design," in *SCSC*, 2007.

[19] P.-A. Mudry, G. Zufferey, and G. Tempesti, "A dynamically constrained genetic algorithm for hardware-software partitioning," in *GECCO*, 2006.

[20] H. Youness, M. Hassan, K. Sakanushi *et al.*, "A high performance algorithm for scheduling and hardware-software partitioning on mpsocs," in *DTIS*, 2009.

[21] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *MICRO*, 2011.

[22] S. Wong, F. Duarte, and S. Vassiliadis, "A hardware cache memcpy accelerator," in *FPT*, 2006.

[23] S. Nilakantan, S. Battle, and M. Hempstead, "Metrics for early-stage modeling of many-accelerator architectures," *Computer Architecture Letters*, vol. PP, no. 99, p. 1, 2012.

[24] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung, "Combining data reuse with data-level parallelization for fpga-targeted hardware compilation: A geometric programming framework," *IEEE TCAD*, 2009.

[25] M. Feng, C. Tian, C. Lin, and R. Gupta, "Dynamic access distance driven cache replacement," *TACO*, 2011.

[26] J. Cong, M. A. Ghodrat *et al.*, "Bin: a buffer-in-nuca scheme for accelerator-rich cmps," in *ISLPED*, 2012.

[27] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *PACT*, 2002.

[28] B. Holland *et al.*, "Rat: Rc amenability test for rapid performance prediction," *TRETS*, 2009.

[29] M. Kim, H. Kim, and C.-K. Luk, "Sd3: A scalable approach to dynamic data-dependence profiling," in *MICRO*, 2010.

[30] G. D. Price, J. Giacomoni, and M. Vachharajani, "Visualizing potential parallelism in sequential programs," in *PACT*, 2008.

[31] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: rethinking and rebooting gprof for the multicore age," in *PLDI*, 2011.