An Idiom-finding Tool for Increasing Productivity of Accelerators

Laura Carrington
UCSD/SDSC
9500 Gilman Dr. MC0505
La Jolla, CA 92093-0505
1-858-534-5063
lcarring@sdsc.edu

Mustafa M. Tikir[1]
Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
1-858-357-1681
mustafa.m.tikir@gmail.com

Catherine Olschanowsky
UCSD/SDSC
9500 Gilman Dr. MC0505
La Jolla, CA 92093-0505
1-858-246-0744
cmills@sdsc.edu

Michael Laurenzano
UCSD/SDSC
9500 Gilman Dr. MC0505
La Jolla, CA 92093-0505
1-858-822-2798
michaell@sdsc.edu

Joshua Peraza
UCSD/SDSC
9500 Gilman Dr. MC0505
La Jolla, CA 92093-0505
1-909-292-6970
jperaza@ucsd.edu

Allan Snavely
UCSD/SDSC
9500 Gilman Dr. MC0505
La Jolla, CA 92093-0505
1-858-534-5158
allans@sdsc.edu

Stephen Poole
ORNL
PO BOX 2008 MS6173
Oak Ridge, TN 37831-6173
1-865-574-9008
spoole@ornl.gov

ABSTRACT

Suppose one is considering purchase of a computer equipped with accelerators. Or suppose one has access to such a computer and is considering porting code to take advantage of the accelerators. Is there a reason to suppose the purchase cost or programmer effort will be worth it? It would be nice to able to estimate the expected improvements in advance of paying money or time. We exhibit an analytical framework and tool-set for providing such estimates: the tools first look for user-defined idioms that are patterns of computation and data access identified in advance as possibly being able to benefit from accelerator hardware. A performance model is then applied to estimate how much faster these idioms would be if they were ported and run on the accelerators, and a recommendation is made as to whether or not each idiom is worth the porting effort to put them on the accelerator and an estimate is provided of what the *overall application speedup* would be if this were done.

As a proof-of-concept we focus our investigations on Gather/Scatter (G/S) operations and means to accelerate these available on the Convey HC-1 which has a special-purpose "personality" for accelerating G/S. We test the methodology on two large-scale HPC applications. The idiom recognizer tool saves weeks of programmer effort compared to having the programmer examine the code visually looking for idioms; performance models save yet more time by rank-ordering the best candidates for porting; and the performance models are accurate, predicting G/S runtime speedup resulting from porting to within 10% of speedup actually achieved. The G/S hardware on the Convey sped up these operations 20x, and the overall impact on total application runtime was to improve it by as much as 21%.

ICS'11, May 31–June 4, 2011, Tucson, Arizona, USA.
Copyright 2011 ACM 978-1-4503-0102-2/11/05…$10.00.

1 This work is completed while Dr. Tikir was an active member of PMaC Labs at SDSC

General Terms

B8.2 Performance Analysis and Design Aids

Descriptors

Performance

Keywords

Benchmarking, performance prediction, performance modeling, FPGAs, accelerators, HPC.

1 INTRODUCTION

Tools to help programmers identify optimization opportunities are useful for improving application scalability [1-4], improving throughput of applications [5], and improving programmer productivity[6, 7]. Lately Scalable hybrid-multi-core computing systems are becoming ubiquitous in the HPC environment. These systems typically have host cores and accelerator hardware thus offering the promise of enhanced compute power. For example the recently announced #1 on the Top500 list augments 14,336 Intel Westmere-EP processors with 7,168 NVIDIA M2050 general purpose GPUs and is capable of 2.57 petaflops on LINPACK. Because some real-world applications are more memory bound than compute bound, other accelerator-based systems such as Convey-HC-1 focus on speeding up memory accesses rather than flops. Yet common wisdom is that all these systems are difficult to program. They require writing code in new language extensions such as CUDA of even (in the case of Convey) coming up with VHDL-level descriptions of the problem to be solved. So at issue is to determine to what extent real-world applications would benefit from the accelerators on such systems? And assuming they would benefit, what portions of the applications would benefit most and how much work would it be to port the application, or portions of it, to these accelerators? PIR (PMaC's Idiom Recognizer) [8] is a static analysis tool that automates the process of identifying sections of code that are candidates for acceleration. PIR automatically recognizes and identifies user-specified compute and memory access patterns, called idioms [41] within application source code. This greatly reduces the amount of code that an expert must analyze "by hand" (visually). Once a section of code is identified that *could* be run on an accelerator, there still remains the question *should* it be? Often the startup overhead of moving the data to/from the accelerator outweighs the performance benefits. Also this question may depend on input. In this work we develop a general performance model for accelerators that can estimate whether the identified idiom would be worth computing on an accelerator depending on input. The combined tool-stream (PIR + model) helps programmers to be productive in two ways 1) it saves them the labor of analyzing thousands lines of legacy code "by hand" to identify idioms that are candidates for acceleration and 2) it saves the time of porting candidate idioms by identifying sections of code/idioms to be ported only when the forecasted performance improvement will benefit overall performance 3) it triages the idioms that should be ported in order best-candidate-for-porting-first.

In this paper we focus our investigations on local Gather/Scatter (G/S) operations and means to accelerate these available on the Convey HC-1 which has a vendor supplied function (i.e. Convey personality) for accelerating local G/S. A local G/S is one where the data is gathered or scattered from memory local to the core and doesn't require communication among cores. G/S is a very difficult memory access pattern for most commodity systems to do well [9-11] and therefore some real applications may benefit more from this kind of acceleration than the more common "flops" accelerators. We test the methodology on two large-scale HPC applications. The idiom recognizer tool saves weeks of programmer effort compared to having the programmer examine the code "by hand" looking for idioms; performance models save yet more time by rank-ordering the idioms, best-candidate-for-porting-first; the models themselves are highly accurate and predict the G/S runtime speedup resulting from going ahead and porting to within 10% of what was actually achieved. The G/S hardware on the Convey sped up these operations 20x, the overall impact on total application runtime was to improve it by as much as 21%. In what follows we describe first in Section II our tool for recognizing idioms, section III describes our performance modeling methodology applied to G/S, section IV provides experimental results, section V concludes, and section VI gives background and related work.

2 IDIOMS

PIR (PMaC Idiom Recognizer)[8] is a tool for searching source code for idioms. An idiom [41] is a local pattern of computation that a user may expect to occur frequently in certain applications. For example, a stream idiom is a pattern where memory is read from an array, some computation may be done on this data, and then the data is written to another array. A stream reads sequentially from the source array and writes sequentially to the destination array. A stream may arise from the presence of the statement A[i] = B[i] within a loop over i.

Idioms are useful for describing patterns of computation that have the potential to be optimized, for example, by loading the piece of code to a coprocessor or accelerator.

The PIR tool allows us to automate searching for idioms in a powerful way by using data-flow analysis to augment the identification process. It would be very difficult to use a simpler searching tool, such as regular expressions, because a regular expression does not naturally discern the meaning of the text it identifies. For example, in the code shown in Figure 1, a simple regular expression based on (for example) "grep" that searches for stream idioms of the form "A[i]= B[i]" would incorrectly identify line 1 as a stream and it would miss the stream at lines 3-4 because the assignment is broken into multiple statements.

```
1. values[c] = constants[c];
2. for( i = 0; i < 10; ++i ) {
3.   item = source_array[i];
4.   dest_array[i] = item;
5. }
```

Figure 1. Sample stream idiom code.

PIR, however, is able to determine that line 1 is not in a loop and that c is a constant. This indicates that the meaning of this statement is simply a variable assignment, rather than a stream. In lines 3-4, PIR uses data-flow analysis to determine that item in line 4 holds a value from the source array making this a stream.

PIR's design provides the flexibility to identify optimization opportunities for many different hardware configurations. The user provides descriptions of the idioms to be identified. As a starting point, PIR provides a set of commonly useful idioms and access to an Idiom definition syntax that allows for user customization of the idioms.

PIR includes seven idiom definitions we have found to be common in HPC applications. The user is free to define more via a simple pattern describing API. The pre-defined idioms are described in the following. All of the code samples are assumed to be part of a loop, i (and j) are loop induction variables.

- Stream: A[i] = A[i] + B[i]

The stream idiom includes accesses that step through arrays. In the above example two arrays are being stepped through simultaneously, but the stream idiom is not limited to this case. Stepping through any array in a loop where the index is determined by a loop induction variable is considered a stream.

- Transpose: A[i][j] = B[j][i]

The transpose idiom involves a matrix transpose, essentially reordering an array using the loop induction variable.

- Gather: A[i] = B[C[i]]

The gather idiom includes gathering data from a potentially random access area in memory to a sequential array. In this example the random accesses are created using an index array, C.

- Scatter: A[B[i]] = C[i]

The scatter idiom is essentially the opposite of gather. Values are read from a sequential area of memory and saved to an area accessed in a potentially random manner.

- Reduction: s = s + A[i]

A reduction can be formed from a stream, as in the working example, or a gather. It implies that the value returned from the read portion of the idiom is assigned to a temporary variable.

- Stencil: A[i] = A[i-1] + A[i+1]

A stencil idiom involves accessing an array in a sequential manner, including a dependency between iterations of the loop.

Table 1 presents just a sample of the report for an application. The sample shows how PIR is able to classify the idiom, capture the source file, source line, function name and even the line number of source code used for the identification(additional information about loop depth, start, and end are captured but not shown).

Table 1. Sample output from PIR analysis on HYCOM.

File Name	Line #	Function	Idiom	Code
mod_tides.F	623	tides_set	gather	pf(i)=f(index(i))
mxkrt.f	992	mxkrtbaj	reduction	sdp=sdp+ssal(k)*q

The PIR user manual and programmers guide can be found online at www.sdsc.edu/pmac.

3 MODELING GATHER-SCATTER OPERATIONS

Once the idioms are identified having an accurate estimate of which ones will perform well on the new accelerator could save a lot of human hours in porting efforts. Some idioms that can be executed on an accelerator should not be because the overhead of moving the data to the accelerator is greater than the performance gains of executing them there. It is not uncommon anecdotally for users to invest a fair amount of time in porting to accelerators only to discover the whole code as a whole runs slower[1]. Having an accurate performance model avoids these situations.

In this work we develop a general methodology to model idiom operations on accelerators. The focus of this paper is on the G/S idiom due to its ability to exacerbate a systems memory performance. The Von Neumann Bottleneck is particularly aggravated by memory access patterns that have a substantial amount of randomness or indirection in the address stream such as Gather/Scatter idioms. In a Gather, non-contiguous memory locations are collected up into a contiguous array; in a Scatter, contiguous array elements are distributed to non-contiguous memory locations; because these species of operations are 1) prevalent in many scientific applications 2) performance-limited on many architecture by the latency of main memory, various architectural features have been proposed to try to accelerate them. An access to main-memory on today's deep-memory-hierarchy machines commonly takes two orders-of-magnitude longer than either floating-point or integer operations, thus these operations will be performance bottlenecks unless some means can be found to accelerate them.

Our motive was to develop models and methodology to be able to assist in the prediction of the benefit of having G/S accelerators directly in future HPC architectures without just building the hardware first and finding out if it is useful afterwards. Building a model of the interaction of the hardware and the application requires three main components: the machine component, the application component, and the model component. The machine component involves measuring the performance benefits of using the acceleration hardware for Gather/Scatter operations and identifying the parameters that affect that performance (i.e. locality, vector length, etc.). The application component entails automating the detection of Gather/Scatter operations in a large scale HPC application and measuring the parameters of these operations that affect performance on the acceleration hardware. The final piece, the model component, combines the machine component and application component to complete the model and detail the performance of the application on the hardware.

3.1 Machine Component- Measuring Gather/Scatter Operations

The Machine Component of the G/S model consists of a way to measure the typical performance of running Gather/Scatter operations on acceleration hardware and determine what parameters affect their performance. A simple benchmark was developed, SGBench[12]. SGBench has two main loop bodies; one for a local scatter operations and one for a local gather operations.

Figure 2 and Figure 3 represents the code snippets from SGBench for the scatter and gather operations respectively. The code represents local operations that do not require communication among cores. Figure 2 illustrates a scatter operation. In this loop the array A is filled by the contents of array B at non-contiguous locations in local memory, determined by the index array. In Figure 3 the gather operation is shown. Here a contiguous piece of array B is filled by the contents of a non-contiguous piece of array A in local memory. In both loops the index array is filled with integers representing elements of an array.

```
for(i=0;i<n;i++){
    A[index[i]] = B[i];
}
```

Figure 2. Loop for Scatter operation.

```
for(i=0;i<n;i++){
    B[i] = A[index[i]];
}
```

Figure 3. Loop for Gather operation.

In considering Gather/Scatter operation and ways to accelerate them, it is important to understand if there is locality in the index set. If there is locality the accelerator hardware may be able exploit it; also accelerator hardware may exist at different levels of the memory hierarchy (cache, local main memory, global memory, etc.). The size of the array accessed then matters but also any clustering or patterns of the index set matters. G/S accelerator hardware basically works by packing and reordering memory requests and pulling in chunks of random locations at a time. So even though the index set is by definition random, if reordered it

[1] Negative results are rarely published in computer science: at a recent DoD GP GPU workshop most application developers reported spending considerable time porting codes to accelerators without getting any speedup.

may have some locality properties that G/S hardware can take advantage of. To study different modes of Gather/Scatter operations, the addresses in the index array in SGBench was varied. This work focused on three specific modes.

Figure 4 depicts the three kinds of locality modes we consider in the index set. Figure 4a *random indices* has no locality, that is to say the index set is a set of entirely random indices that span the entire array from 0 to size of A. The second mode, *clustered indices*, shown in Figure 4b, has locality clusters within the random index set. In this case the indices in the index array span certain sections of the A array and within those sections the indices of the index array are random. In the third mode, *spread indices*, illustrated in Figure 4c, the indices have, if reordered, some spatial locality (predictable or constant strides) spanning the entire array from 0 to size of A.

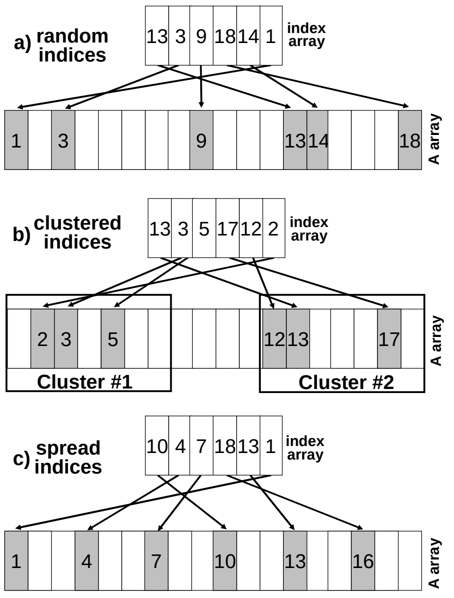

Figure 4. Type of index arrays for gather-scatter operations, a) random index, b) clustered index, c) spread index.

Random indices (Figure 3a) might be typical of a graph problem or similar to the RandomAccess (GUPs) kernel [13, 14], Clustered (Figure 3b) is typical of sorting partially-sorted input, Spread (Figure 3c) typical of a sparse matrix problem which can arise in Finite Element or Finite Difference codes. Along with enabling the index array to be filled in the three different modes, SGBench also allows the user to vary the padding or offset between the A, B and index arrays as these parameters may interact with memory banking.

3.1.1 Machine Component – measuring the FPGAs

To study the performance effects of Gather/Scatter operations on acceleration hardware, SGBench was ported to the Convey HC-1[15]. The Convey HC-1, shown in Figure 5, uses a tightly integrated Intel 5138 processor (Xeon Woodcrest) with a FPGA-based, reconfigurable coprocessor. The coprocessor can be targeted at specific workloads by reloading it with different instruction sets, called personalities. By enabling the implementation of a new instruction set, the coprocessor can be tailored to specific applications and algorithms. In addition the coprocessor shares memory with the Intel processor, which reduces the data transfer time between the computing elements and eliminates much implementation complexity.

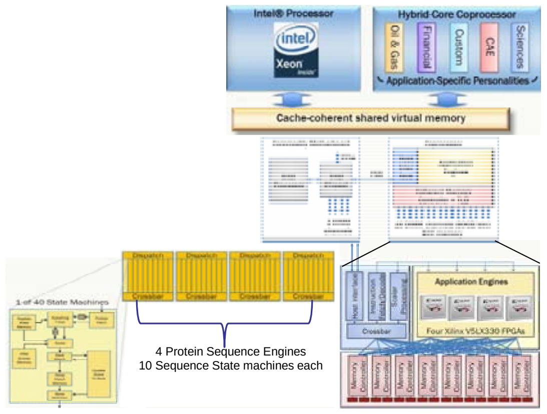

Figure 5. Convey HC-1.

For this work, a Convey supplied personality was used to accelerate local Gather/Scatter operations illustrated in Figure 2 and Figure 3. This personality was used both to gather performance data used as input for the model and to port sections of the application for model verification.

The SGBench benchmark was used to measure both the performance of Scatter operation and Gather operation. SGBench was run on two ways; first the entire SGBench execution was run on the host processors of the HC-1. Second the majority of the SGBench execution was run on the host processor with just the loops containing Gather or Scatter operations running on the FPGAs. The measurements were taken to determine the performance effects of running G/S operations at increasing data set sizes (i.e. total address range of the arrays). The measurements were made using an index array of stride-1 and an index array of random-stride, this was intended to cover the range of performance for the operations shown in Figure 4a through Figure 4c. The measurements were taken both on the host Xeon processor and the FPGA coprocessor. Figure 6 and Figure 7 illustrate the results of these measurements as a function of the size of the address range of the test loop and compare performance of operations on the host Xeon processor with those on the FPGAs.

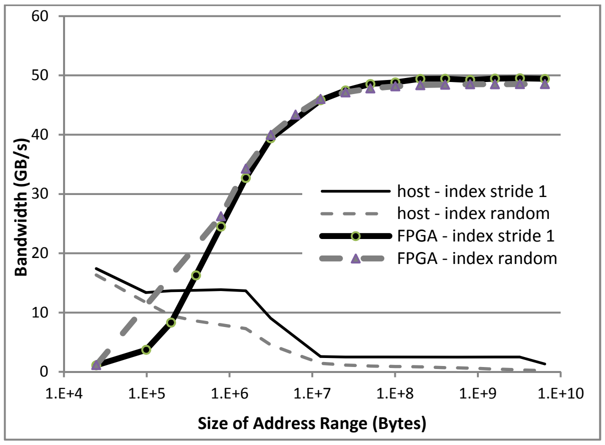

Figure 6. Performance of Scatter loop as a function of |Address Range| on Convey HC-1 using host and FPGA.

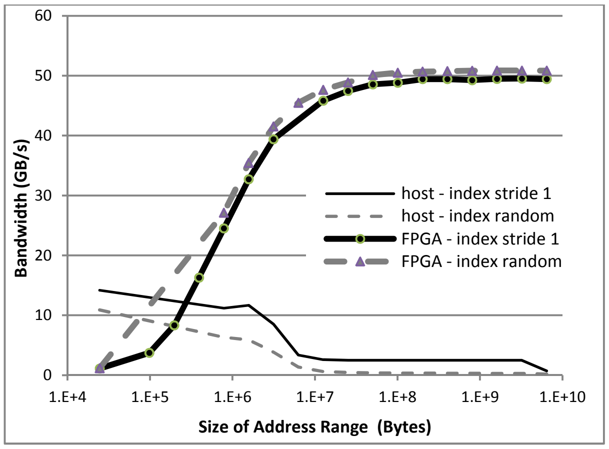

Figure 7. Performance of Gather loop as a function of |Address Range| on Convey HC-1 using host and FPGA.

Figure 6 shows Scatter operations run at different data set sizes. The first curve is for SGBench run at increasing sizes on the host processor (Xeon) of the Convey HC-1 with an index array of stride-1, while the second curve is for an index of random-stride. These curves illustrate that as the size is increased the performance decreases in a stepwise fashion on the host processor an exemplar of today's deep-memory-hierarchy machines comprised of levels of cache. The next two curves, in Figure 6, are Scatter operations run on the FPGA with stride-1 and random-stride index arrays. This illustrates that at a few small data set sizes, it is more beneficial to run Scatter operations on the host but that performance on the FPGA *increases* with size asymptotically and for large sizes ~200x performance improvement over the host can be gained. More importantly, the performance does not change much when using random-stride vs. stride-1 index array on the FPGA for larger data sizes.

Figure 7 represents similar measurements for Gather operations. Both figures illustrate that there are only small changes in performance when running Gather vs. Scatter operations on the FPGAs and the FPGA's performance is not significantly affected by the randomness of the index array. The figures illustrate that the data footprint of these operations can dramatically affect performance; in other words there is no such thing as "the performance of a machine on Gather/Scatter" rather one needs more information such as the size of the range of array address arguments to accurately estimate performance.

3.2 Application Component

In order to have a general scheme for modeling and predicting Gather/Scatter operation performance, we first need to identify instances of Gather/Scatter operations within the source code. Secondly, we need to capture the parameters of each individual G/S instance that will affect their performance (locate them on the benchmark graph) if they were to be ported to the accelerated hardware. As identified above, size or range of addresses in the G/S loop, are important in determining performance on the Convey HC-1 and thus were identified as the main modeling parameter. Figure 6 and Figure 7 illustrates how the Gather/Scatter operations benefit for large address ranges more when using the FPGAs, which would also likely be true of commodity (HPC) processors with built-in G/S capabilities. In fact in retrospect we can look back at the vector systems of the 80' and 90's and see how this was also true back then. Back then G/S was inherent in the hardware and on the Crays it was in the ISA as an assembler vector instruction.

3.2.1 Identifying Gather/Scatter operations

Once the basic Gather/Scatter operation is defined the next step is to automate the detection of these operations in large scientific application because without automated detection the task of identifying candidates for G/S acceleration by hand would be extremely time consuming. The PIR tool, described in section 2, was used to automate the search for G/S instances in large scale scientific application.

Identification of the idioms in an application in an automated way allows for the easy detection of Gather/Scatter instances. Once the G/S instances are detected the second step is capturing the address ranges for each instance. This needs to be done dynamically by instrumentation.

3.2.2 Measuring range Gather/Scatter operations

Once operations in an application are identified as Gather/Scatter, the next step, as suggested by the benchmark results, is to measure the address range of each instance as they occur within the application. It should be clear that the range may depend on input, thus a static analysis tools such as PIR, while sufficient to *identify* instances of Gather/Scatter in applications, provides insufficient information to accurately model them since performance may vary more than an order-of-magnitude just depending on range of addresses (see Figure 6 and Figure 7). Therefore we used the binary instrumentation tool PmacInst[16] to instrument the instances of Gather/Scatter identified by PIR.

PmacInst is used to gather the memory traces of an application for the general PMaC performance model. It is designed to instrument an identified set of basic-blocks in an application and capture the memory addresses of those blocks during the execution of the application. To conserve time and space the address stream is simulated against architectural features of interest (caches, Gather/Scatter hardware) on-the-fly while the application is running.

In order to capture the range of the Gather/Scatter operations identified by PIR two steps were required. First the information gathered from PIR needed to be translated and conveyed between the two tools because PIR works on source code while PmacInst works on the binary—we need to identify the binary code corresponding to the source code. This translation allows PmacInst to add additional instrumentation to those basic-blocks identified by PIR. Secondly, an address range function was developed to process the address streams from Gather/Scatter operations and calculate the range and distribution of those operations.

In order to translate identified PIR operations to basic-blocks by PmacInst a special feature of PmacInst was utilized. This feature allowed the source file and line number to be collected along with the basic-block number. This same information is collected by PIR, so by using additional parsing scripts, the PIR output was connected to and combined with the static analysis from PmacInst to automate the identification and additional instrumentation of all Gather/Scatter instances. This extra instrumentation allowed for the addresses of these operations to be processed by the function to calculate address range for each G/S instance.

To calculate the range of addresses in a Gather/Scatter operation the function was designed to minimize overhead while maintaining sufficient accuracy by determining the memory regions touched by each basic-block. One way to gather information on memory regions for an address stream is to use binary search tree that holds the boundary addresses for memory regions and at every memory access searches for the region the memory access fits in [17]. This requires additional split and merge operations of memory regions according to some heuristics for accurate region identification. Even though such an approach would potentially identify the memory regions very accurately, it would also introduce a significant overhead since a search in the binary tree would be required for every memory access and would rely on accurate split and merge heuristics.

To avoid this large overhead, the function tracks the addresses accessed by each memory operation (instruction) in the basic-block separately. Since the instrumentation code already passes information about the memory operation for each access, we identify the region accessed by the memory operation by keeping track of the minimum and maximum addresses touched.

This method is faster at instrumentation time but requires additional post processing to be accurate enough. The overall overhead required for this additional instrumentation is less than 10%. The additional post-processing is needed due to the fact that even though the list of memory regions accessed by all memory operations can be used as the memory regions accessed by the block, some of these memory regions may overlap. This is can be a result of multiple memory operations accessing the same data structures and arrays, which can be an outcome of heavy code optimizations such as loop unrolling. To correct for such overlap, we post process the trace data to find the minimal number of memory regions accessed by a basic block. We accomplish this by first sorting the list of memory regions accessed by all memory operations in ascending order of their minimum addresses and then merging the overlapping regions.

Table 2 shows the results of using this range calculation on an instrumented run of the SGBench benchmark. The table presents the measured and actual ranges for a given array in the scatter loop shown in Figure 2. The instrumented SGBench was run at 8 different size scatter loops and the results for 4 of those are shown in Table 2 below. The relative absolute error for all the runs was less than 1% for all sizes.

Table 2. Actual and measured ranges in SGBench.

Array name	Actual size (bytes)	Measured size (bytes)	% Error[2]
A	20,480	20,400	0.4
B	5,120	5,080	0.8
Index	2,560	2,540	0.8
A	65,536	65,520	0.0
B	16,384	16,344	0.3
Index	8,192	8,192	0.3
A	16,777,216	16,777,128	0.0
B	4,194,304	4,194,264	0.0
Index	2,097,152	2,097,132	0.0
A	33,554,432	33,554,352	0.0
B	8,388,608	8,388,568	0.0
Index	4,194,304	4,194,284	0.0

[1] $\% \text{ Error} = abs\left(\frac{\text{actual size - measured size}}{\text{actual size}}\right)\times 100$

The automated process of using PIR to identify Gather/Scatter operations and PmacInst to measure ranges was further tested using spot checking of two large scale applications (HYCOM and Flash) and showed similar absolute relative error with all loops measured with less than 1% error.

Thus by combining the PIR and PmacInst tools we have a tool to automate the process of first identifying Gather/Scatter operations and second measuring the range of each of those operations in a tractable fashion, which is an important step in modeling their performance on acceleration hardware.

3.3 Modeling Gather/Scatter operations

We extended our existing performance modeling framework to account for the performance effects of acceleration hardware on Gather/Scatter operations. Here we briefly describe the PMaC framework designed to model large scale parallel applications and then present how the Gather/Scatter model is incorporated into the framework.

3.3.1 PMaC Performance prediction framework

The PMaC prediction framework is designed to accurately model parallel applications on HPC systems. In order to model a parallel application, the framework is composed of two models, a computational model and a communication model. The computational model models work done on the processor in between communication event, while the communication model deals with modeling communication events. Below a brief description is provided but for a detailed description of the framework, please see Snavely et al.[18], Carrington et al.[19] and Tikir et al. [20].

For each model, the computational and communication is comprised of three primary components: an application signature, a machine profile, and a convolution method. The machine profile captures that rates that a machine can perform fundamental operations through simple benchmarks. These simple benchmarks includes tests for performance of different kinds of memory access patterns, arithmetic operations, and communications events, at various working set sizes and message sizes. The application signature includes detailed information about the required operations the applications needs as well as the locality of its data and its message sizes, and is collected via trace tools. The machine profile and application signature are combined by mapping the required operation of the application to their expected rate on the target machine. This mapping takes place in the PSiNS simulator that re-plays the entire execution of the HPC application on the target/predicted system to calculate the runtime of the target system. The models generated by the framework have shown good accuracy (i.e. <15% absolute relative error) predicting full-scale application running production datasets on existing systems [21].

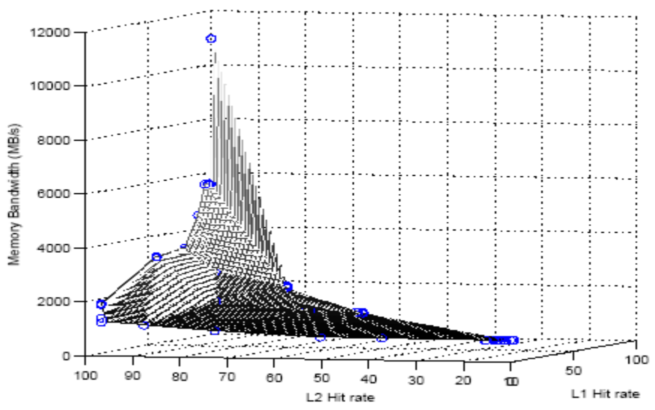

Figure 8. . Measured bandwidth as function of cache hit rates for Opteron.

For this work we focused on extending the computational or core model, which is comprised of the work done on the processor or core in between communication events or computational work. We extended the model to account for the effect of having accelerators in each node; and modeling the work that is off-loaded to the accelerator.

For the computational model there are two main operations that normally comprise a majority of the run time: arithmetic time and memory time. Arithmetic time is the time required to perform floating-point and other math operations. Memory time is the time required to load and store memory references and it is this time that usually dominates the computational model's run time. To accurately model memory time we need to determine the number of bytes that need to be loaded or stored but also the location and access pattern of those references. This is because references from different locations and access patterns can perform orders of magnitude better or worse. For example, a stride-one load from L1 cache can perform significantly faster than a random stride load from main memory. Figure 8 is an example of the MultiMAPS benchmark[20] used to capture the complex interaction of spatial and temporal locality and performance on memory reference on a two level Opteron processor. The MultiMAPS surface allows the model to determine the performance of memory references with different spatial and temporal locality.

To capture this temporal and spatial locality information in an application for each reference that occurs in the execution an augmented memory trace of the application is used. The memory trace is captured using the pmacInst[16] tool which has the ability to instrument each memory reference in order to capture the memory address stream from each core of the running application. This address stream is then processed on-the-fly through a cache simulator for the target system or system to predict. The result of this trace is for each basic block of the application information is provided on: 1) the location of the block in the source code, 2) number of floating-point operations and type, 3) number of memory references and type (e.g. load and/or store), 4) size of references in bytes, and 5) the expected cache hit rates for those references on the target system. It is the hit rates that provide information about the spatial and temporal locality of the reference and enable accurate predictions of their performance via the corresponding data on the MultiMAPS surface. The framework enables the application to be traced on a one system (i.e. the base system) but by simulating a difference cache structure (i.e. the target system) the model can predict the performance of a completely different system. For this work the applications were trace on an IBM Power 6 system but simulated the cache structures for AMD and Intel based systems, these are referred to cross-architectural predictions.

In the computational model the majority of the time comes from the memory time or time to move data through the memory hierarchy (arithmetic time is also modeled but memory time tends to dominate in the cases we studied). A detailed description of the memory time calculation can be found at Tikir et al.[20]. The general memory time equation is:

1) $$\text{memory time} = \sum_{i}^{\text{all BB}} \frac{(\text{memory ref.}_{i,j} \times \text{size of reference})}{\text{memory BW}_j}$$

Where:

Memory bandwidth$_j$ =	memory bandwidth of the jth type of memory reference on a target system.
Size of reference =	size in bytes of the reference
# memory ref.$_{i,j}$ =	number of memory references for basic-block i of the jth type (locality information encompassed in type)

Equation 1 represents the memory time of an application as being the sum of all basic-block's memory time for the application.

To model Gather/Scatter operations, an additional equation needed to be developed to compute the time for operations off-loaded to the accelerator or Convey HC-1 FPGAs and additions to the simulator on which equation is used for which basic-block.

3.3.2 Gather/Scatter memory model

In order to add the modeling of Gather/Scatter operations the simulator requires two additional pieces of data. First, how the performance of Gather/Scatter operations vary with size of operation and second which basic-blocks in the application contains these operations and what is their size.

In developing a model of the performance of Gather/Scatter operation based on size, we first investigated the trends displayed in Figure 6. The concept in designing the model is to start as simple as possible and only add complexity when needed. Therefore we started with a piece-wise fit to the data in Figure 6. The equation for modeling Gather/Scatter operations on the FPGAs is as follows:

2) $$\text{memory time} = \sum_{i}^{\text{all BB}} \frac{(\#\,\text{memory ref.}_{i,j} \times \text{size of reference})}{\text{memory BW}_{gs}}$$

Where:

3) $$\text{memory BW}_{gs} = 1E-5 \times \text{size} + 12.6 \text{ for } 192\text{KB} < \text{size} < 3\text{MB}$$

4) $$\text{memory BW}_{gs} = 48 \geq 3\text{MB}$$

Where:

Memory BW$_j$ =	memory bandwidth of the jth type of memory reference on the target system.

Size = size in bytes of the reference

Equation 2 represents the memory bandwidth as a function of size in bytes detailed in Equations 3 and 4. The framework was modified so that the simulator incorporated the trace data of the PIR recognized Gather/Scatter operations, its measured size, and Equations 2, 3, and 4 to model the performance impact the Convey would have on applications. Note that the simulator includes a conditional that would only run those Gather/Scatter operations that are predicted to see a performance improvement on the FPGA (from Figure 6 and Figure 7 only those with array address ranges larger than 192 KB).

4 EXPERIMENTAL MODEL AND RESULTS

In the experiment we used two full-scale scientific applications to test the model, HYCOM and FLASH. FLASH is an astrophysics application developed at the "FLASH Center" funded by DOE ASC/Alliance Program. It is a state-of-the-art simulator code for solving nuclear astrophysical problems related to exploding stars. It is 126,478 lines of code. HYCOM is an ocean modeling application developed by The Naval Research Laboratory (NRL), Los Alamos National Laboratory (LANL), and the University of Miami as an upgrade to MICOM (well-known ocean modeling code) by enhancing the vertical layer definitions within the model to better capture the underlying science. It is 54,085 lines of code.

The main goal of the research was to design a modeling framework to explore the question "Would a given HPC application benefit from accelerators?". We designed a general modeling methodology around this but to verify the methodology we chose a specific set of operations (e.g. G/S) and a specific accelerator (e.g. the FPGA). In order to verify the models two main experiments were conducted. The first experiment was to simply verify the model for the two large scale applications running on a large scale HPC system. In addition verify the model at a finer-grain level for only Gather/Scatter operations still running on the same system, see section 4.1. The second experiment dealt with a smaller data set for the FLASH application. In this experiment we wanted to verify the FPGA Gather/Scatter model by running the application on the Convey HC-1 and porting a single loop of the application to the FPGA for the verification of the FPGA model. The smaller data set was used because the HC-1 only had 4 host cores and we were unable to run a bigger data set. Figure 2 and Figure 3 had already verified the accuracy of the FPGA Gather/Scatter model on benchmarks on the HC-1 but this experiment verified the model working within a full-scale application on that system.

4.1 Models and fine grain verification experiment

In this experiment we needed to first verify the full application model for the two applications running on a large scale HPC system. In addition verify the accuracy of the full application model at a finer-grain level by capturing the model time for only Gather/Scatter operations of the application. To accomplish this a large data set was used as input to the application. For this experiment we identified all of the Gather/Scatter instances within both applications, modeled the entire computational time of the applications (using the full framework), then inside the simulator captured the modeled time for just the Gather/Scatter idioms within the applications, and verified those modeled times on a large scale HPC system by measuring those times with a light weight profiling tool and individual timers.

For this experiment, models were developed for HYCOM and FLASH for Oakridge National Laboratory's (ORNL) Jaguar XT4 to verify the accuracy and the granularity of the models. Jaguar is an XT4 with seastar2 interconnect [22]. FLASH was run on Jaguar with 128 processors using the white dwarf input and HYCOM was run with 59 processors using the 26-layer 1/4 degree fully global HYCOM benchmark input deck. While the framework is designed and is used at much larger processor counts, both applications were run at small core counts to minimize the contribution of runtime from communication, thus focusing the modeling on computational aspects of the application such as their ability to benefit from Gather/Scatter accelerators.

PIR was used to capture the number and location of Gather/Scatter operations in FLASH and HYCOM source codes. An additional feature was added so that PIR automatically inserted tags at the identified Gather/Scatter operations to enable timer insertion to aid in model accuracy verification. This feature could further be enhanced to automate the process of modifying the identified Gather/Scatter loops to call the FPGA personality, a valuable aid in the porting process (future work).

After the PIR analysis, 140 Gather/Scatter idioms were identified for FLASH and 64 in HYCOM. PIR only captures the number of G/S idioms but gives no indication on their contribution to the overall runtime; for that, the performance model is required. A performance model for FLASH and HYCOM executing on Jaguar was developed to investigate the performance benefit these applications would see from FPGA accelerators.

Table 3 shows the results for both predicted/modeled and measured computational time of FLASH and further breaking down the time spent in Gather/Scatter operations for both predicted/model and measured time. It also has the predicted/modeled computational time on Jaguar for HYCOM compared to the measured computational time.

Table 3. Prediction of FLASH on ORNL Jaguar.

Code segment	Predicted time (sec)	Measured time (sec)	% error[1]
FLASH – full	262	250	4.6%
FLASH G/S ops	69	68	1.4%
HYCOM – full	5956	5781	2.9%

[1] $\%\ \text{Error} = abs\left(\frac{\text{measured - predicted}}{\text{measured}}\right) \times 100$

Table 3 shows that the accuracy of modeling FLASH and HYCOM on existing hardware is accurate to within 5% (e.g. rows 2 and 4). Additionally at finer granularity models of the Gather/Scatter operations of FLASH with the same level of accuracy (e.g. <2% error, row 3). Due to the nature of the HYCOM G/S operations (e.g. inner loops with other operations) we were unable to capture the fine grain G/S operation time alone without disturbing the overall runtime significantly due to number of calls to the timer routines (the measurement was affecting the execution significantly). Therefore we were only able to compare the overall runtime and not the G/S operations on HYCOM.

4.2 Convey HC-1 models and verification

Once the accuracy of the applications' models on an existing system was confirmed then the exploration of the Gather/Scatter operations on the Convey system was investigated. It is one thing to verify the accuracy of a kernel extracted from an application (e.g. SGBench) but this experiment was designed to verify the accuracy of the model during execution of the entire application.

To verify the accuracy of the FPGA Gather/Scatter model and simulation for large scale applications, one of the identified Gather/Scatter operations (i.e. loop) of FLASH was ported to the Convey system and timed to compare the simulated FPGA model time with the measured time. So a majority of the FLASH application could then be executed on the HC-1 host processors with one loop being executed on the FPGA. Porting loops to the Convey requires some additional work; the whole point of our technique is to focus programmer efforts on the parts of the code where the effort will result in the most reward. From FLASH we chose porting a loop that ranked the highest among the Gather/Scatter operations for overall runtime contribution.

Due to the size of the Convey HC-1 we were accessing (only 4 host processors) we had to choose a smaller FLASH input, sedov-2d problem rather than the white dwarf input, in order to run FLASH at full-scale with the single ported loop executing on the FPGAs. Alternately we could have extracted the loop into a small kernel but that might not fully capture the data transfer penalties associated with the many visits to the loop throughout the execution. A full application model was developed for the sedov-2d input similar to the white dwarf input verified in Table 3. This time the model was developed for not only Jaguar but the host processors of the HC-1. In addition for both models the simulation time for the identified loop was captured in order to verify the model accuracy at the loop level. Then a model was generated for the HC-1 with that loop executing on the FPGA, this allowed verification of the FPGA G/S model. The sedov-2d input computational time was predicted/modeled and measured on Jaguar and the Convey host; the results are shown in Table 4 below. This illustrates the accuracy of the Gather/Scatter model on the Convey HC-1 system for the sedov-2d input with a relative absolute error less than 10%.

Table 4. Simulated and measured Gather/Scatter times for FLASH on the Convey HC-1.

Code/section (system)	Measured time (s)	Predicted time (s)	% Error[1]
FLASH-full (Jaguar)	518.6	487.7	6.0%
#1 FLASH-G/S loop (Jaguar)	3.4	3.7	9.3%
FLASH-full (Convey Host)	491.9	489.7	5.0%
#1 FLASH-G/S loop (Convey FPGAs)	2.8	3.0	8.6%

[1] $\% \text{ Error} = abs\left(\frac{\text{measured - predicted}}{\text{measured}}\right) \times 100$

Table 4 verifies the accuracy of the FPGA G/S model when implemented in a full-scale application even at a fine grain loop level with only 8.6% abs. relative error. Table 3 and Table 4 confirms the accuracy of the full application models for FLASH and HYCOM, the accuracy of the models on a finer-grain level (i.e. gather/scatter operations in FLASH), and the accuracy of the FPGA Gather/Scatter model on the Convey HC-1 FPGAs.

4.3 Exploring the benefits of G/S on FPGAs

In section 4.1 we verified the Jaguar models accuracy for full scale applications and sub-sections (e.g. G/S loops) of the application. In section 4.2 we verified the FPGA G/S model executing in the context of a full scale application. After verifying these components of the modeling framework we then began to investigate the original question the methodology was designed to explore, "Do applications benefit from running G/S operations on FPGAs?" Since we have verified large scale runs of HYCOM and FLASH on Jaguar nodes we will start our exploration there.

To explore this space we need to create a hypothetical system consisting of 32 Jaguar nodes (e.g. 128 cores) with FPGAs attached to the cores, essentially a hypothetical Convey system where the host processors are Barcelona (i.e. XT4- Jaguar) processors rather than Xeon's. We then use the models to predict the performance of FLASH and HYCOM on this system to answers the question "What if all Gather/Scatter operation that would perform faster on the FPGAs were ported and run on them how that would affect overall application runtime?"

These new simulations give insight into not only whether an application might benefit from accelerated G/S operations but how much and which ones. The G/S operations in FLASH took 68 seconds on Jaguar and were predicted to take 3.5 seconds if run on the FPGAs an almost 20X speedup. HYCOM showed slightly different behavior since a significant number of the HYCOM G/S operations did not benefit from the FPGAs (were too small). HYCOM showed G/S operations predicted to run on the FPGAs at a 7X speed up compared to Jaguar. The overall runtime speed up of FLASH and HYCOM resulting from the FPGAs on the new hypothetical system was 21% and 3.2% respectively.

The model showed that while there are over 140 Gather/Scatter operations in FLASH that contributes to 27% of the runtime, the FPGAs can potentially speedup these by close to 20X. The model allows users to focus porting efforts by identifying only those applications and idioms that would benefit most. While the speedup of the G/S operations was significant, due to the nature of the applications this speedup had only a modest contribution to the speedup of the overall runtime. This speedup could potentially be significantly improved if other idioms were ported to the FPGAs (future work).

If we look at Figure 6 we see that G/S operating on arrays smaller than 16K would be faster on the host. The performance models predict that the 33 largest instances of G/S (out of 140) correspond to 95% of predicted total G/S execution time. Quantitatively, the predicted time with FPGA G/S model is 197.2 seconds if all G/S in FLASH larger than 16K are ported while porting the top 33 results in execution time of 198.7 seconds for accelerator system. That is, even though we eliminated 107 blocks from porting to G/S hardware, we did not lose anything from the benefits of the accelerators. We only lost 1.42 seconds of optimized time but we were able to reduce the port time by a significant factor.

5 CONCLUSIONS

This work showed a general modeling methodology to automate the prediction of HPC applications on acceleration hardware. The models were confirmed using two large scale applications with an average absolute relative error of 5% and fine-grain accuracy of the model was confirmed with similar results. The fine-grain model for FPGA G/S operations was proven to have less than 8.6% absolute relative error using a loop from FLASH running on the FPGAs for verification. The performance modeling methodology estimated that >100 instances of the G/S idiom were not worth porting thus saving additional programmer's time and improving performance and avoiding illogical results such as incorrectly assuming Convey can't accelerate G/S because blind porting all G/S would make the code run slower. While speedup of 21% and 3.2% may not seem significant for some developers this amount may be worth the effort. The tools and models make the porting effort less challenging by identifying which sections of the application would benefit from porting and indicating their contribution to the overall runtime, leaving the final decision up to the developer. In addition, this work focused only on Gather/Scatter operations but the tools and methodology can be applied to other operations (i.e. stream, reduction, etc.). And with additional work on PIR and its source code tagging feature, porting could become quite effortless for the user/developer. In future work we also intend to extend this to model other idioms and also in using these potential calculations to predict how much energy we could save (FPGA's consume less energy than their host processors). Also, the methodology described in this work could easily be extended to other accelerators such as GPUs. Such an extension would involve similar steps as using PIR to identify code blocks which might perform efficiently on GPUs, developing a model for those blocks on the GPU, develop trace tools to capture relevant model inputs, and modify the simulator to incorporate new trace data and models. Such work is saved for future work.

6 Background

The prevalence of Gather/Scatter operation in application can be seen in sorting algorithms, hash searches, and sparse-matrix vector multiplication[23] to name a few. For many parallel algorithms scatter and gather are two fundamental operations[24] for instance radix sort is a parallel sorting algorithm[25], hash used in databases, and any linear solvers use sparse-matrix vector multiplication[26]. The use of acceleration hardware to speed-up Gather/Scatter operations has mainly focused on GPU-based acceleration hardware. In He et al[23] they used gather scatter operations optimize using GPUs to implement three memory intensive algorithms radix sort, the hash search, and the sparse-matrix vector multiplication with models for just the GPU.

On traditional architectures there are varying techniques for modeling the performance of HPC applications [18, 27-46], spanning derived analytical models, trace-based models, to a combination of the two. Analytical based models require a detailed understanding of the application and/or its algorithm and the method doesn't lend itself to automated model generation, unlike trace-based methods.

Understanding the performance of acceleration hardware through modeling is a task that many researchers have focused on. Alam et al[47] investigate using their Modeling Assertions to model the multi-streaming, vector processing capabilities of the X1E on the NAS SP kernel[48]. Hong and Kim[49] developed an analytical model for GPU performance and applied it to micro-kernel and benchmarks, but not full scale HPC applications. Govindaraju et al [50] developed a memory model for GPUs for a set of algorithms used in scientific applications. They tested the model on benchmark kernels but not full-scale HPC applications.

This work offers a unique contribution in that it develops a general framework to model acceleration hardware on full-scale HPC applications and validates the model using full-scale applications and the acceleration hardware offered by the Convey FPGA system.

Acknowledgment

This work was supported by the DoD and used elements at the Extreme Scale Systems Center, located at ORNL and funded by the DoD. The software used in this work was in part developed by the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago. This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Special thanks to Mark Kelly and Glen Edwards for all their help.

References

[1] B. Miller, *et al.*, "The Paradyn Parallel Performance Measurement Tool," *Computer*, vol. 28, pp. 37-46, 2002.

[2] S. Shende and A. Maloney, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, 2006.

[3] V. Adve, *et al.*, "An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs," *Proceedings of the IEEE/ACM SC95 Conference*, 1995.

[4] V. Freeh, *et al.*, "Analyzing the Energy-time Trade-off in High-Performance Computing Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, pp. 835-848, 2007.

[5] J. Shin, *et al.*, "Autotuning and Specialization: Speeding up Nek5000 with Compiler Technology," presented at the International Conference on Supercomputing, 2010.

[6] J. Kepner, "HPC Productivity: An Overarching View," *International Journal of High Performance Computing Applications*, vol. 18, 2004.

[7] L. Hochstein, *et al.*, "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers," *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[8] C. Olschanowsky, *et al.*, "PIR: A Static Idiom Recognizer," in *First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010)*, San Diego, CA, 2010.

[9] J. Nieplocha, *et al.*, "Global Arrays: A Non-uniform Memory Access Programming Model for High-Performance Computers," *Journal of Supercomputing*, vol. 10, pp. 169-189, 1996.

[10] J. Lewis and H. Simon, "The Impact of Hardware Gather/Scatter On Sparse Gaussian Elimination," *SIAM J. Sci. Stat. Comput.*, vol. 9, pp. 304-311, 1988.

[11] S. Mukherjee, *et al.*, "Efficient Support for Irregular Applications on Distributed-memory Machines," *ACM SIGPLAN Notices*, vol. 30, pp. 68-79, 1995.

[12] *SGBench see, http://www.sdsc.edu/pmac/SGBench*.

[13] J. Dongarra and P. Luszczek, "Introduction to the HPC Challenge Benchmark Suite," ICL-UT-05-01, 2005.

[14] G. Fox, *et al.*, "Solving Problems on Concurrent Processors: Volume 1, Chapter 22," P. Hall, Ed., ed Englewood Cluffs, NJ, 1988.

[15] C. HC-1, "http://www.conveycomputer.com/ConveyArchitectureWhiteP.pdf," ed.

[16] M. Tikir, *et al.*, "The PMaC Binary Instrumentation Library for PowerPC," *Workshop on Binary Instrumentation and Applications, San Jose,* 2006.

[17] C. Olschanowsky, *et al.*, "PSnAP: Accurate Synthetic Address Streams Through Memory Profiles," *The 22nd International Workshop on Languages and Compilers for Parallel Computing*, Oct. 8-10 2009.

[18] A. Snavely, *et al.*, "A Framework for Application Performance Modeling and Prediction," *ACM/IEEE Conference on High Performance Networking and Computing*, 2002.

[19] L. Carrington, *et al.*, "How well can simple metrics represent the performance of HPC applications?," *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing*, 2005.

[20] M. Tikir, *et al.*, "Genetic Algorithm Approach to Modeling the Performance of Memory-bound Codes," *The Proceeding of the ACM/IEEE Conference on High Performance Networking and Computing*, 2007.

[21] M. Tikir, *et al.*, "PSINS: An Open Source Event Tracer and Execution Simulator for model prediction," presented at the HPCMP User Group Conference, San Diego, CA, 2009.

[22] "ORNL Jaguar see http://www.nccs.gov/computing-resources/jaguar/."

[23] B. He, *et al.*, "Efficient Gather and Scatter Operations on Graphics Processors," *SC07*, 2007.

[24] J. D. Owens, *et al.*, "A Survey of general purpose compuation on graphics hardware," *Computer Graphics Forum*, vol. 26, 2007.

[25] M. Zagha and G. E. Blelloch, "Radix sort for vector multiprocessors.," in *Supercomputing 1991*, 1991.

[26] J. Bolz, *et al.*, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," *ACM Transactions on Graphics*, pp. 917-924, 2003.

[27] V. Adve and R. Sakellariou, "Application representations for multiparadigm performance modeling of large-scale parallel scientific codes," *The International Journal of High Performance Computing Applications*, vol. 14, 2000.

[28] S. Alam and J. Vetter, "A Framework to Develop Symbolic Performance Models of Parallel Applications," presented at the 5th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems, 2006.

[29] G. Almasi, *et al.*, "Demonstrating the scalability of a molecular dynamics application on a Petaflop computer," presented at the Proceedings of the 15th international conference on Supercomputing, Sorrento, Italy, 2001.

[30] B. Armstrong and R. Eigenmann, "Performance forecasting: Towards a methodology for characterizing large computationals applications," in *Internationals Conference on Parallel Processing*, 1998.

[31] D. Bailey and A. Snavely, "Performance Modeling: Understanding the Present and Predicting the Future," *EuroPar*, 2005.

[32] J. Bourgeois and F. Spies, "Performance prediction of an NAS benchmark program with chronosmix enviroment," presented at the 6th International Euro-Par Conference, 2000.

[33] M. Clement and M. Quinn, "Automated performance prediction for scalable parallel computing," *Parallel Computing*, vol. 23, 1997.

[34] M. J. Clement and M. J. Quinn, "Analytical performance prediction on multicomputers," *Supercomputing*, pp. 886-894, 1993.

[35] D. Culler, *et al.*, "LogP: Towards a realistic modle of parallel computation," in *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[36] M. Faerman, *et al.*, "Adaptive performance prediction for distributed data-intensive applications," presented at the Supercomputing, 1999.

[37] T. Fahringer and M. Zima, "A static parameter based performance prediction tool for parallel programs," presented at the The International Conference on Supercomputing, 1993.

[38] D. J. Kerbyson, *et al.*, "Predictive Performance and Scalability Modeling of Large-Scale Application," *Supercomputing*, 2001.

[39] C. Lim, *et al.*, "Implementation lessons of performance prediction tool for parallel conservative simulation," presented at the 6th International Euro-Par Conference, 2000.

[40] G. Marin and J. Mellor-Crummey, "Cross Architecture Performance Predictions for Scientific Applications Using Parameterized Models," *In Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, June 2004.

[41] B. Mohr and F. Wolf, "KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications," presented at the European Converence on Parallel Computing (EuroPar), 2003.

[42] J. Simon and J.-M. Wierum, "Accurate Performance Prediction for Massively Parallel Systems and its Applications," *Euro-Par'96 Parallel Processing*, vol. 1124, pp. 675-688, 1996.

[43] A. van Gemund, "Symbolic performance modeling of parallel systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, 2003.

[44] A. Wagner, *et al.*, "Performance models for the processor farm paradigm," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, 1997.

[45] L. Yang, *et al.*, "Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution," presented at the Proceedings of the 2005 ACM/IEEE conference on Supercomputing, 2005.

[46] X. Zhang and Z. Xu, "Multiprocessor Scalability Predictions Through Detailed Program Execution Analysis," *International Conference on Supercomputing*, pp. 97-106, 1995.

[47] S. Alam, *et al.*, "An Exploration of Performance Attributes for Symbolic Modeling of Emerging Processing Devices," presented at the HPCC, 2007.

[48] *NAS Parallel Benchmarks (NPB) see, http://www.nas.nasa.gov/Resources/Software/npb.html*.

[49] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," presented at the ISCA'09, Austin, Texas, USA, 2009.

[50] N. Govindaraju, *et al.*, "A Memory Model for Scientific Algorithms on Graphics Processors," presented at the Supercomputing, Tampa, Florida USA, 2006.