UNIVERSITY OF CALIFORNIA, SAN DIEGO

HPC Application Address Stream Compression, Replay and Scaling

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Catherine Rose Mills Olschanowsky

2011

The dissertation of Catherine Rose Mills Olschanowsky is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Chair

University of California, San Diego

2011

DEDICATION

To Roman.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to that my advisor Dr. Allan Snavely for his support and guidance throughout my graduate career. I would also like to thank the rest of my committee Dr. Michael Norman, Dr. Nicholas Schork, and especially Dr. Tajana Simunic Rosing and Dr. Scott Baden for their insight and help.

I owe a special debt of gratitude to my family, especially Roman, Konrad and Danny; they have been with me every step of the way. I would not have been successful without the guidance of Dr. Michelle Strout, my mentor, best friend and big sister, thank you. Thank you also to everyone at SDSC and especially the PMaC lab who have been so supportive. I would like to thank specifically Kate Kaya and Shava Smallen for their kind support.

Chapter 3 contains material reprinted from the proceedings of the Workshops on Languages and Compilers for Parallel Computing. Olschanowsky, C.M. and Tikir, M.M. and Carrington, L. and Snavely, A. *PSnAP: Accurate Synthetic Address Streams Through Memory Profiles.* 2009. [46]. The dissertation author was the primary investigator and author for this paper.

Chapter 3 contains material reprinted from a submission to the IEEE International Symposium on Workload Characterization. Olschanowsky, C.M. and Carrington, L. and Snavely, A. *A Tool for Characterizing and Succinctly Representing the Data Access Patterns of Applications.* 2011. [43]. The dissertation author was the primary investigator and author for this paper.

Chapter 6 contains material reprinted from the dissertation authors research exam for the Department of Computer Science and Engineering, University of California, San Diego. *Simulation guided performance modeling.* 2008. [45] The dissertation author was the sole author of this paper.

This work was supported in part by the DOE Office of Science through the SciDAC2 award entitled Performance Engineering Research Institute.

This work was part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing

Applications, IBM, and the Great Lakes Consortium for Petascale Computation.

This research was supported in part by the National Science Foundation through the San Diego Supercomputer Center and utilized the Dash and Triton resources.

VITA

2001	B. S. in Computer Science *cum laude*, University of California, San Diego
2004	M. S. in Computer Science, University of California, San Diego
2011	Ph.D. in Computer Science, University of California, San Diego

PUBLICATIONS

Olschanowsky C., L. Carrington, A.Snavely. A Tool for Characterizing and Succinctly Representing the Data Access Patterns of Applications. Submitted to *IEEE International Symposium on Workload Characterization*, 2011.

Olschanowsky C., L. Carrington, M. Laurenzano, M. Tikir, T. Simunic-Rosing, and A. Snavely. On-Node Application Energy Consumption Modeling. In: *The DOD High Performance Computing Modernization Office Users Group Conference*, June 2010.

Olschanowsky C., M. Meswani, L. Carrington, and A. Snavely. PIR: PMaC's Idiom Recognizer. In: *The International Workshop on Parallel Software Tools and Tool Infrastructure (PSTI)*, September 2010.

Olschanowsky C., M. Tikir, L. Carrington, and A. Snavely. PSnAP: Accurate Synthetic Address Streams Through Memory Profiles. In: *The* 22^{nd} *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, October 2009.

Wright N., S. Smallen, C. Olschanowsky, J. Hayes, and A. Snavely. Measuring and Understanding Variations in Benchmark Performance. In: *The DOD High Performance Computing Modernization Office Users Group Conference*, June 2010.

Kahali O., J. He, C. Olschanowsky, H. Casanova, and A. Snavely. Measuring the Performance and Reliability of Production Computational Grids. In: *The* 7^{th} *IEEE/ACM International Conference on Grid Computing*, September 2006.

Smallen S., C. Olschanowsky, K. Ericson, P. Beckman, and J. Schopf. The Inca Test Harness and Reporting Framework. In: *The International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2004.

ABSTRACT OF THE DISSERTATION

HPC Application Address Stream Compression, Replay and Scaling

by

Catherine Rose Mills Olschanowsky

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Professor Allan Snavely, Chair
Professor Scott Baden, Co-Chair

As the capabilities of high performance computing (HPC) resources have grown over the last decades, a performance gap has developed and expanded between the processor and memory. Processor speeds have improved according to Moore's law, while memory bandwidth has lagged behind. The performance bottleneck created by this gap, termed the "Von Neuman bottleneck," has been the driving force behind the development of modern memory subsystems.

Many advances have been made aimed at hiding this memory bottleneck. Multi-level cache structures with a variety of implementation policies have been introduced. Memory subsystems have become very complex and the effectiveness of their structure and policies vary according the the behavior of the application running on the resource.

Memory simulation studies aid in the design of memory subsystems and in acquisition decisions. During a typical acquisition, candidate resources are evaluated to determine their appropriateness for a pre-defined workload. Simulation-aided models provide performance predictions when the hardware is not available for full testing ahead of purchase. However, address streams of full applications may be too large for direct use, complicating memory subsystem simulation.

Memory address streams are extremely large. They can grow at a rate of over 2.6 TB/hour per core. HPC workloads contain applications that run for days across hundreds of processors, generating address streams whose handling is intractable. However, the memory address streams contain a wealth of information about the behavior of applications, that is largely inaccessible.

This work describes a novel compression technique, specifically designed to make the information within HPC application address streams accessible and manageable. This compression method has several advantages over previous methods: extremely high compression rates, low overhead, and a human readable format. These attributes of the compression technique enable further, previously problematic, studies.

High compression ratios are a necessity for application address streams. Address streams are very large, making them challenging to collect and store. Furthermore, any simulation experiment performed using the stream will be limited by disk speeds, since there is no other plausible place to store and retrieve such volumes of data. The compression technique presented has demonstrated compression ratios in the hundreds of thousands of times. This leads to file sizes that can easily be emailed between collaborators and the format can be replayed at least as fast as disk speeds.

The collection overhead for an address stream must be low. The collection takes place on an HPC resource, and HPC resource time is costly. This compression technique has an unsampled average slowdown of 90X. This slowdown is an improvement of the state-of-the-art.

The compressed address stream profiles are human readable. This attribute

enables new and interesting uses of application address streams. It is possible to experiment with hypothetical code optimizations using simulation or other metrics rather than actually implement the optimizations.

Strong scaling analysis of memory behavior is historically challenging. High-level metrics such as execution time and cache miss rates do not lend well to strong scaling studies because they hide the true complexity of the application-machine interactions. This work includes a strong scaling analysis in order to demonstrate the advanced capabilities that can be built upon this compression technique.

Chapter 1

Introduction

1.1 High-Performance Computing

High-Performance Computing (HPC) systems perform large-scale scientific calculations and simulations that require computational power well beyond the abilities of a single processor. HPC systems have evolved to accommodate the growing demands of scientific applications. HPC systems, also termed supercomputers, were originally very fast scalar processors. Vector processors were introduced in the 1970's and multiple processors running in parallel became the norm in the 1990's. Today HPC systems include each of these evolutionary steps, many multiprocessors working in parallel each of which contains a vector processing unit. Many HPC systems even include specialized GPUs and FPGAs.

Many of the innovations in computer architecture have been aimed at hiding the latency of memory operations. The "Von Neumann bottleneck" is a term used to describe growing gap between floating point operation performance and memory bandwidth. The rate at which floating point operations can be executed has increased according to Moore's law for the last 3 decades. Memory bandwidth has not kept pace. This makes memory bandwidth the limiting factor for most HPC applications.

HPC applications are typically data-intensive. A taxonomy of scientific calculations, referred to as the seven motifs, was suggested by Colella [14] in order to better define HPC computational and data-movement requirements. The categories are dense linear algebra, sparse linear algebra, spectral methods, N-body methods, structured grids, unstructured grids and Monte Carlo. Most of these motifs are by nature, data-intensive.

Exascale computing is the next big step for HPC systems and memory bandwidth continues to be a major hurdle. The DOE Exascale Computing Study [18] names four primary challenges to achieving exascale computing. One of the primary challenges is in data storage and movement, specifically emphasizing the challenges to performance caused by waiting for data.

Understanding and manipulating the data-movement behaviors of HPC applications is a requirement for achieving Exascale performance for real applications. The data access patterns of applications must be readily available for study and optimization. This work focuses on making HPC data access patterns available and demonstrates the types of studies that are enables by the gained accessibility.

1.2 Memory Performance

Due to the effect of the "Von Neumann" bottleneck, the memory performance of an HPC application often dominates overall per-core performance to the point that it can alone be used as an estimate of total on-node performance. The per-core performance of a data-intensive HPC application is, therefore, a combination of the application's memory access patterns and the processor's memory subsystem.

Memory access patterns are embodied in the application's memory address stream. An application's memory address stream is the collection of memory addresses requested by the application during execution. Once capture the address stream can be used in a variety of studies, most commonly, memory simulation studies. However, memory address streams are difficult to collect and use, because of their size and the large time overhead associate with their collection.

A typical memory subsystem includes several levels of cache, the translation look-aside buffer (TLB) and main memory. A typical processor has two to three levels of cache, each level may be shared between data and the instructions or used exclusively by one. The smallest (and fastest) caches are those closest to the processor; the caches may also be shared among cores. The TLB speeds up virtual to physical address translations. Main memory is the largest and slowest memory storage.

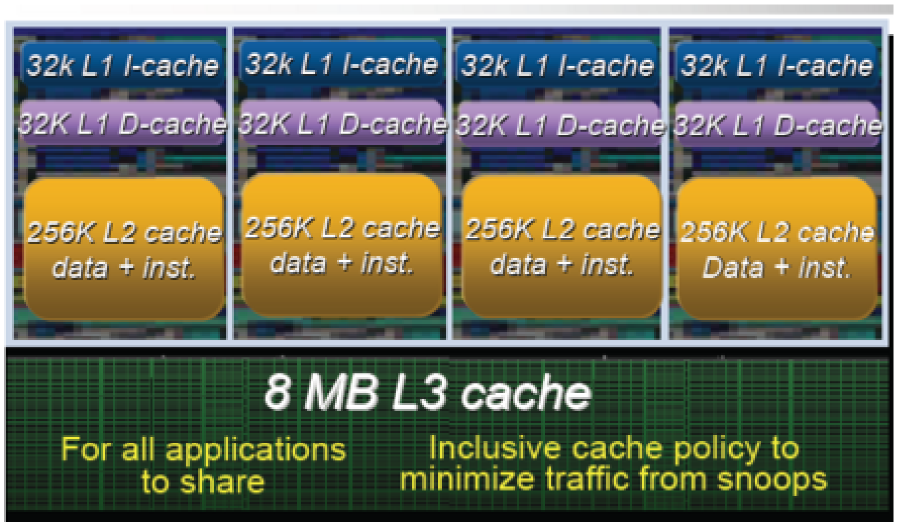

32k L1 I-cache
32K L1 D-cache
256K L2 cache data + inst.
32k L1 I-cache
32K L1 D-cache
256K L2 cache data + inst.
32k L1 I-cache
32K L1 D-cache
256K L2 cache data + inst.
32k L1 I-cache
32K L1 D-cache
256K L2 cache Data + inst.
8 MB L3 cache
For all applications to share
Inclusive cache policy to minimize traffic from snoops

Figure 1.1: The cache structure of Intel's Nehalem Architecture (image courtesy of Tom's Hardware).

Figure 1.1 shows the cache hierarchy present in Intel's Nehalem architecture. A full description of the hierarchy includes not only the sizes of the included caches, but also the replacement policies, inclusion policies, the associativity and many other descriptors. This high level of complexity is one of the reasons that tuning an application to work for a specific architecture can be difficult. It is also a motivating factor behind the use of simulation for memory performance modeling.

The interactions between the application and resource can become quite complex. The resource implements policies that are meant to improve performance in general, but are not application specific. For instance, many processors implement prefetching from various levels of cache. Ineffective prefetching can lead to wasted power consumption and possibly performance degradation.

The processor and memory subsystem can be represented with a fixed model. Attributes include processor cycle speed, cache sizes, associativity and replacement policies, bandwidths and latencies.These attributes remain constant for a single processor.

The application's memory access attributes are potentially more complicated and while the processor doesn't change (for a particular machine) the application variations are endless. We are seeking some fairly compact and complete representation of that infinitely variable behavior that is embodied in the address stream, but we lack the space to store and the time to evaluate in detail the stream in its entirety. Therefore, we are looking for patterns and "shorthand" to capture the important properties.

1.2.1 Trace-Driven Memory Simulation

Trace-driven memory simulation is the main consumer of collected and synthetically created address streams. It is applicable to system design and evaluation, compilation (via trace-driven optimizations), and performance tuning. Today it is a standard practice to use address traces to explore the memory behavior of applications [52, 37, 12, 4, 3].

Simulation allows for the evaluation of new memory hierarchy designs without hardware implementation and this benefits both system design and evaluation for procurement. Modeling current workloads on proposed systems via simulation provides valuable insights, aiding in procurement decisions[55, 36]. Compiler optimization choices can be guided and evaluated through the examination and simulation of the resulting address streams. The accuracy and usefulness of each of these applications depends directly on the availability of relevant input, specifically relevant address traces.

Using address streams from an actual scientific workload is the best policy for achieving accurate performance predictions and evaluations. The validity of a simulation depends heavily on the chosen input workload; in the case of memory simulations the input is an address trace [19, 29]. VanderWiel [64] points out in a comparison study of two prefetching techniques, the performance improvement varied widely for each workload complicating the choice of prefetching technique. However the collection and handling of such memory address streams proves to be very challenging.

1.2.2 Challenges to the Collection and Storage of Memory Address Streams

Due to the growth in the size of memory traces the direct collection and storage of the traces is no longer tractable [23]. 2.6 TB an hour per core is a conservative estimate of the address stream growth rate. Collecting an address trace for an application that runs several hours on thousands of processors is therefore not reasonable unless one leverages some regularity or recurring patterns in the application [34]. Traces that can be replayed by a cycle accurate simulator include instructions as well and are therefore even larger.

In order to circumvent the collection and handling of full application traces several approaches have been evaluated. At one extreme, some have suggested using a random number generator to generate a stream of addresses, but it has been shown that selecting address streams that closely represent the behaviors of an existing workload is very important for an accurate examination of the memory subsystem[54]. Much effort has been devoted to finding alternate means to generate realistic synthetic address streams[24, 57, 33, 65]. Achieving within 90% verisimilitude when using these streams as representatives of the full application to predict cache hit rates has not been possible until recently. Recently Weinberg [67] introduced the idea of profiling address streams and storing descriptions of them, which could later be used to generate realistic address streams for simulation. That work circumvents the problems related to collecting and storing full address traces, but does not maintain fine-grained patterns within the stream.

Compression techniques have been explored as another approach. Two recent projects are Sequitur [39] and Path Grammar Guided Trace Compression (PGGTC) [21]. Each of these represent major improvements upon previous attempts, but are not satisfactory as complete solutions. The main drawback of each is the time overhead required to create of the compressed format. Perhaps more important than the overhead is the inaccessibility of the format. Neither presented a human readable format that enabled directly study and manipulation of the patterns, meaning that a person could gain understanding of the access patterns by reading the compressed format.

The direct collection and storage of address streams has remained problematic, yet the need for access to full application address streams is growing. As HPC resources grow to include many more processors application scalability, improving per-node memory performance, becomes even more important.

1.3 Application Scalability

In addition to the on-node performance that is dominated by memory performance application scalability during strong scaling is greatly affected by memory performance. Increased performance is pursued along two main routes; increasing the computing power available to applications by adding processing units (scalability) and tuning applications to squeeze every bit of performance out of specific architectures (serial efficiency). Until now we have been describing how the memory address stream is key to exploring serial efficiency, but it is also the key to understanding per-core scaling behavior.

Understanding and predicting the scaling performance of scientific applications on large high performance computing(HPC) resources is critical to the design and utilization of future larger HPC resources [7]. In order to properly reason about the scaling performance of an application it is necessary to thoroughly understand the changes in the memory behavior of the application during scaling. Although communication and computation time both contribute to the overall scaling behavior of scientific applications, computation time dominates execution time in many modern scientific applications. For instance, the SPECFEM [32] code is a scientific application that is being tuned to run at very high processor counts. After recent optimizations only 5.9% of the overall execution time was attributed to blocking communication time when SPECFEM was run across 16K cores [13]. That means that during 94% of the execution time computation was taking place and therefore, the scaling behavior of the computation phase is the dominating factor in the behavior of the overall application. Many of the scientific applications running on today's high performance computing resources are memory bound [61], meaning that the movement of data from memory to registers dominates the running time.

Modeling and predicting the changes in memory behavior as an application scales leads to running (computation) time predictions, which are key to understanding application scaling.

New systems will be composed of a larger number of processors than today's and may have different and more complex memory subsystems. In order to evaluate how current applications will perform on future resources we must understand and model the application's memory behavior as it scales. In data-intensive computing the time spent moving data to and from memory dominates the on-processor execution time [55].

Application scaling is divided into two large categories: strong and weak scaling. Strong scaling holds the overall problem size constant and adds more processors to the computation. Weak scaling holds the problem size on each processor steady; the overall problem size and the number of processors available increase proportionally. In reality strong and weak scaling are two extremes in a spectrum. Strong scaling has a greater effect on the on-processor execution time while weak scaling holds the per-processor computation fairly constant while adding more communication. This makes predicting the performance with respect to on-processor execution a key factor for predicting strong scaling. This work focuses directly on predicting the strong scaling behavior of an application with respect to the memory hierarchy. That is, it focuses on analyzing the temporal and spatial locality characteristics of an application in order to predict its strong scaling behavior.

Strong scaling is generally understood to be the harder of the two types to predict. As the data is spread across larger numbers of cores, the access patterns and working set sizes change. This creates seemingly sudden and unpredictable jumps in performance as the application is scaled. Super-linear speedup can be observed when a working set suddenly fits into a smaller level of cache. At the same time, as the data per core shrinks, the ratio of the data that comprises the surface, and therefore data that is likely to be shared increases. This may cause areas of the code that are insignificant at low core counts to become the dominant areas at high core counts.

1.4 Thesis Statement

A compressed representation of an application address stream that comprises stride patterns of individual constituent instructions provides tangible advantages over the state-of-the-art. The representation is more compact than previous techniques, incurs lower collection overhead, can be replayed at speeds limited only by the bandwidth of the storage media, is easily human-readable, and reproduces the performance attributes of the original stream with 99% fidelity as measured by the cache miss rate. In addition, the representation can be modified to perform experiments such as reasoning about the scaling behaviors of the application.

1.5 Dissertation Structure and Contributions

The remainder of this dissertation is organized as follows. Chapter 3 provides an overview of a novel address stream compression technique aimed toward well-structured loops. The contributions of this work are the compression technique, which provides very high compression rates, low collection overhead, and a human readable format.

Chapter 4 demonstrates the use of the compression technique and its accessible format for a strong scaling study. The contribution of this work is a method for identifying patterns during scaling at low core counts that can be extrapolated out to larger core counts.

Chapter 5 describes an approach to grouping the address streams from parallel executions into groups. Only a single representative from each group needs to be saved. The contributions of this work are that it reduces the data size of the saved address streams even further, and enables a full simulation based performance model to be created during a scaling study.

Chapter 2

Benchmarks and Applications

The experiments presented in the following chapters are performed on a set of benchmarks and applications. The benchmarks and applications were chosen to demonstrate the effectiveness of our approach on computations common to scientific applications. Each of the benchmarks and applications is described below.

2.0.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks are a benchmark suite written to represent the major types of computations performed by high performance computing applications [8]. The following highlights the important aspects of each.

CG a conjugate gradient method used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication, using unstructured matrix-vector multiplication.

EP an "embarrassingly parallel" kernel, which evaluates an integral by means of pseudo-random trials. This kernel, in contrast to others in the list, requires virtually no interprocessor communication.

FT a three-dimensional partial differential equation solution using FFTS. This kernel performs the essence of many "spectral" codes. It is a rigorous test of long-distance communication performance.

LU a regular-sparse, block (5 X 5) lower and upper triangular system solution. This problem represents the computations associated with the implicit operator of a newer class of implicit CFD algorithms. This problem exhibits a somewhat limited amount of parallelism.

IS a large integer sort. This kernel performs a sorting operation that is important in "particle method" codes. It tests both integer computation speed and communication performance.

MG a simplified multi-grid kernel. This requires highly structured long-distance communication and tests both short- and long-distance data communication.

SP solves a synthetic system of nonlinear PDEs using a scalar pentadiagonal solver kernel.

Alterations to the NAS Parallel Benchmarks

The NPBs were altered to change the memory allocation approach. This alteration was necessary to ease the comparison of profiles taken across core counts. This section outlines the change that was made and documents the performance impact incurred.

The NPBs are written such that the dataset size and core count it is to be run on are known at compile time. This means that for each dataset size and core count combination a recompile is necessary. The profiler names each basic block in the executable. The recompile is problematic for the profiler, because each executable may have small differences that result in inconsistencies in basic block naming across traces.

The changes to prevent a recompile are quite simple. Each benchmark was updated to allocate the memory needed per core after the MPI initialization call. This allows the benchmark to dynamically determine the number of cores available for execution and determine the data size per core.

A small performance impact was observed due to this change. Table 2.1 shows the execution times averaged over three runs. The IS benchmark is not included, because it cannot be run at over 1024 cores and it is not used in the scaling study. The BT and SP benchmarks incur the largest impact (12 and 10 seconds respectively at a core count of four). However, the impact on most of the benchmarks is statistically insignificant. It is expected that the execution times can vary around 5% under normal conditions. At the higher core counts the impact is much smaller, even for BT and SP.

Table 2.1: Performance Variation Due to Dynamic Data Allocation

Bench Mark	**Average Execution Time (seconds)**									
	Original					**Altered**				
	4	16	64	256	1024	4	16	64	256	1024
BT.B	114.4	33.0	9.4	3.2	1.6	126.3	36.8	10.1	3.3	2.1
CG.B	37.2	11.7	5.3	3.1	2.0	36.6	11.8	5.2	3.1	2.1
EP.B	18.3	4.7	1.2	0.4	0.1	18.4	4.9	1.3	0.4	0.1
FT.B	23.6	9.4	3.3	2.2	0.6	23.3	9.4	3.3	2.1	0.7
LU.B	74.3	23.5	6.6	3.0	N/A	78.0	27.8	8.1	3.0	N/A
MG.B	2.2	0.8	0.2	0.1	0.1	2.1	0.8	0.2	0.1	0.1
SP.B	110.6	37.0	8.8	4.1	2.8	120.3	39.6	9.3	4.1	3.0

2.0.2 Avus

AVUS(Air Vehicles Unstructured Solver) [1] from AFRL/VAAC is a finite volume unstructured-grid Euler/Navier-Stokes solver, and derived from Cobalt-60 by the same group. The fundamental algorithm of AVUS is the finite-volume, cell-centered, first-order accurate in space and time, exact Riemann solver of Godunov. AVUS can treat two-dimensional, axi-symmetric, and three-dimensional problems. The grid can be composed of cells of arbitrary type, i.e., tetrahedra, quadrilaterals, pyramids, triangles, etc. Different cell types are permitted within the same grid. The set of boundaries forming each cell, called faces, can also be arbitrary (triangles, pentagons, lines, etc.), though each cell boundary face should be convex. Further, the grids may be decomposed into subdomains, called groups or zones, permitting parallel processing where each zone resides on a separate processor. [1]

2.0.3 Lammps

LAMMPS [47] is a classical molecular dynamics code that models an ensemble of particles in a liquid, solid, or gaseous state. It can model atomic, polymeric, biological, metallic, granular, and coarse-grained systems using a variety of force fields and boundary conditions. In the most general sense, LAMMPS integrates Newton's equations of motion for collections of atoms, molecules, or macroscopic particles that interact via short- or long-range forces with a variety of initial and/or boundary conditions. For computational efficiency LAMMPS uses neighbor lists to keep track of nearby particles. The lists are optimized for systems with particles that are repulsive at short distances, so that the local density of particles never becomes too large. On parallel machines, LAMMPS uses spatial-decomposition techniques to partition the simulation domain into small 3d sub-domains, one of which is assigned to each processor. Processors communicate and store "ghost" atom information for atoms that border their sub-domain. LAMMPS is most efficient (in a parallel sense) for systems whose particles fill a 3d rectangular box with roughly uniform density. [2]

2.0.4 PFLOTRAN

PFLOTRAN [25] is a numerical modeling ground-water model application. Numerical modeling is a critical tool to the U.S. Department of Energy for evaluating the environmental impact of remediation strategies for subsurface legacy waste sites. Unfortunately, the physical and chemical complexity of many sites overwhelms the capabilities of even most state of the art ground-water models. Of particular concern is the representation of highly-heterogeneous stratified rock/soil layers in the subsurface and the biological and geochemical interactions of chemical species within multiple fluid phases. There is clearly a need for higher-resolution modeling (i.e. increased spatial and temporal resolution) and increasingly mecha-

[1]Text From: http://www.erdc.hpc.mil/hardSoft/Software/avus
[2]Text From: http://lammps.sandia.gov

nistic descriptions of subsurface physiochemical processes (i.e. increased chemical degrees of freedom). We present SciDAC-funded research being performed in furthering the development of PFLOTRAN, a parallel multiphase flow and multicomponent reactive transport model. Written in Fortran90, PFLOTRAN is founded upon PETSc data structures and solvers. We are employing PFLOTRAN to simulate uranium transport at the Hanford 300 Area, a contaminated site of major concern to the Department of Energy, the State of Washington, and other government agencies. By leveraging the billions of degrees of freedom available through high-performance computation using tens of thousands of processors, we can better characterize the release of uranium into ground-water and its subsequent transport to the Columbia River, and thereby better understand and evaluate the effectiveness of various proposed remediation strategies. [3]

[3]Text From: [25]

Chapter 3

Address Stream Compression

3.1 Introduction

Obtaining and storing relevant address traces is a fundamental requirement for trace-driven memory simulation of large-scale HPC applications. Workload modeling, which depends heavily on trace-driven simulation, informs procurement decisions [55, 36] by estimating existing workload performance on proposed new resources. However, system evaluation via memory simulation is not limited to procurement; new hardware and software system features, such as prefetching policies, can also be evaluated [64] using trace-driven simulation.

Additionally, power consumption patterns have recently been shown to have a very high correlation with memory access patterns [44] opening up another pivotal area of research dependent on address stream traces. Data movement (or the performance of data movement) and power consumption have been identified as a primary challenge to exascale computing [18]. This means that the need for access to memory address streams is intensifying.

Despite the general usefulness of trace-driven memory simulation, the collection, handling and storage of memory address traces remains problematic. Address streams for large-scale high performance computing (HPC) applications are more problematic than most due to large numbers of processors, acting as a multiplier to address stream trace size, and scarcity of resources, HPC systems are in high demand and few users are willing to consume allocation on performance tuning.

The main challenges associated with address stream collection include space costs, time costs, accessibility, and proxy inaccuracies.

Space Costs: Address streams are extremely large. It is possible for an application's address stream to grow faster than 2.6 TB per hour per core [23]. HPC applications are run on thousands of processors and potentially for many hours, further exacerbating the issue. As a small example, an address stream recorded from the NAS Parallel Benchmarks [8] exceeded a terabyte; these are short running benchmarks, in this case run on only four cores using the smallest HPC dataset(A).

Time Costs: Collecting address streams and storing them to disk, as well as retrieving them from disk, is slow. The processing rate for an address stream that is being read from disk is limited by disk speed. Optimistically, this would be 250 MB/s (addresses can be generated faster than 700 MB/s).

On-the-fly processing of address streams bypasses the space challenges by never storing the trace. However, this processing must be done on an HPC resource and causes at least a 10X slowdown even with aggressive sampling techniques. Without sampling, the slowdown is much worse, 100X-1000X [20]. Additionally, the process must be repeated any time the experimental parameters change, requiring further use of the HPC resource.

Accessibility: Raw address streams themselves provide little if any insight to code structure or access patterns. A stored address stream is too large to be read and understood by a person. The address traces themselves do not provide any insight into the structure or fine-grained patterns created by the application that could inform tuning decisions.

Inaccuracies: One common approach to avoiding the use of large address streams is to work with application kernels or representative benchmarks, rather than real workloads. However, it is very difficult to find a benchmark that can act as an accurate proxy for HPC applications. The validity of a simulation depends heavily on the chosen input workload [19, 29]. VanderWiel [64] points out in a comparison study of two prefetching techniques, the performance improvement varied widely for each workload. This complicates the evaluation of hardware and policy changes. The performance results obtained by traces of small benchmarks chosen to represent a high performance computing (HPC) workload are of questionable relevance during such evaluations; choosing appropriate benchmarks is a difficult task, especially when applied to an HPC workload [40].

Obtaining and storing relevant address traces is a fundamental requirement for trace-driven memory simulation of large-scale HPC applications and the question must be asked: *how does one provide valid and relevant input of substantial size to a simulation?*

Methods such as trace compression, truncation, on-the-fly processing, and synthetic trace generation have each been explored in response to this question. Each of these solutions has shortcomings. Compression techniques incur a large slowdown [39, 22], and some of them require that the entire trace be stored before being compressed [39]. Truncating the trace loses valuable information. On-the-fly processing is done successfully, but uses a large amount of time on valuable HPC resources and has to be rerun each time the evaluation study changes [55]. Previous synthetic trace generation approaches have not reached high enough levels of accuracy [57, 24].

This work presents PMaC's[1] Synthetic Address Stream Profiles, (PSnAP). PSnAP is a lossy compression technique designed specifically for HPC address streams. It takes advantage of patterns found in the per instruction address streams of an application and creates very small profiles. These profiles can be used to replay the address stream. PSnAP resolves many of the difficulties associated with address stream collection including space costs, time costs, accessibility, and proxy inaccuracies).

Space Costs: PSnAP profiles are extremely compact, in the range of kilobytes, meaning they can be emailed between collaborators. Not only are the profiles small, but they do not grow as a function of running time. The size of a PSnAP profile is determined by the complexity of the application code. A very simple, but long-running loop will have the same sized profile as the same loop run over a short time. This is very meaningful for HPC applications, because they often

[1]PMaC is the Performance Modeling and Characterization Lab at the San Diego Supercomputer Center.

perform the same operations repeatedly on different data.

Time Costs: PSnAP profiles can be reused repeatedly without the use of an HPC system. The slowdown incurred by tracing for PSnAP is only incurred a single time and tracing can be performed for specific areas of an application, further reducing HPC system use. The slowdown associated with PSnAP (approximately 100X) is small when compared to any on-the-fly processing method that does not use aggressive sampling. The replay times for PSnAP are fast enough that they can be replayed at least as fast as disk speeds. Some traces are more complex and time-intensive for replay and we present a replay solution that essentially takes up more space than the original PSnAP profiles, but allows for replay at disk speeds.

Accessibility: The PSnAP profiles are human readable and manipulatable. High level structures and fine-grained patterns can be seen in the profiles, application phases can be identified, and the results of compiler optimizations such as loop-unrolling and function inlining can be seen. It is also possible to change the profiles by hand in order to evaluate hypothetical optimizations.

Proxy Inaccuracies: There is no need to use benchmark kernels to represent HPC application address streams; the actual streams can be compressed. We demonstrate that the synthetic address streams generated by PSnAP result in cache miss rates within 1% of the observed rates. We also demonstrate visually that we are able to match not only the statistical properties of the stream, but also the fine-grained patterns.

The contributions of this work are as follows:

- A new per-instruction recording technique that captures access patterns in a concise manner.
- A human readable address stream profile that can be manipulated to evaluate different auto-tuning strategies and reveals address stream metrics that lend themselves to extrapolation for scaling studies.
- A synthetic stream generation method that leverages existing control flow compression and per-instruction address streams.

We evaluate PSnAP based on accuracy and performance using the NAS

Parallel Benchmarks [8]. Accuracy must be examined because PSnAP uses lossy techniques. We compare the observed and synthetic address streams using cache miss rates. The cache miss rates are compared over the entire benchmark execution as well as on a per basic block basis. PSnAP achieves accuracy in the full execution measurements, on average the difference between observed and synthetic streams is .08%.

The evaluation of PSnAP's performance includes: execution overhead, profile size and replay time. A comparison of these values shows that PSnAP beats the state of the art in all three categories. We also demonstrate one of the primary advantages of PSnAP. The profile sizes do not grow as a function of time, but as a function of pattern complexity.

In addition to good performance, the format of PSnAP profiles allows for an array of new usage models. For example, PSnAP profiles can be partially replayed for examination of specific loops, can be replayed in chunks in order to fit into memory, and can be manipulated for experimentation.

3.2 Overview of Approach

The goal of PSnAP is to produce address stream profiles that are small enough to easily store, share and process, while at the same time, contain enough information to create a synthetic address stream that very closely resembles the original. PSnAP, therefore, comprises three steps. First, the capture phase, during PSnAP capture the original address stream is observed, and pertinent information is recorded. Second, the post-processing phase, during PSnAP post-processing the raw PSnAP profile is reformatted in order to create a more accessible and quickly processed format. Third, the replay phase, during PSnAP replay the post-processed profile is read and a synthetic address stream is created.

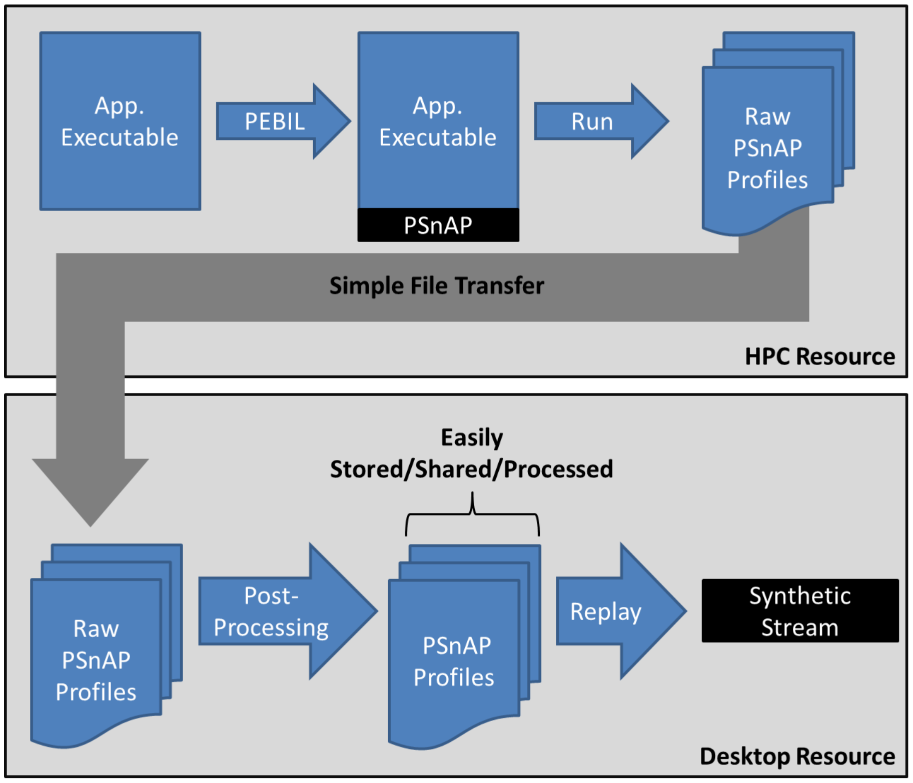


Figure 3.1: Only the first step of the PSnAP process requires the use of an HPC resource.

Each PSnAP stage has targeted goals, each contributing to the overall PSnAP goals. The capture phase is designed to summarize the application's address stream with minimal memory and time overhead. Capture is performed using a binary rewriting tool, PEBIL [35].

The post-processing phase is a simple classification and formatting task. Each memory instruction results in a section in the raw profile. The instruction data is parsed and classified in a way that makes replay simple and fast. This processing phase is light-weight and can be performed on a desktop machine, it does not require the use of an HPC resource.

The replay phase, also does not require the use of an HPC resource. The goal of the replay phase is to be fast, flexible and portable. Not only can the stream be replayed in its entirety, but replay of single basic blocks, loops or nested structures is also possible.

3.3 PSnAP Capture

PSnAP is designed to achieve a high-level of accuracy while maintaining low overhead costs in both time and space. Attempts at doing this in the past have demonstrated the benefit of treating the address stream as many parts rather than holistically [46]. When viewed as a whole, address streams are complex and patterns are difficult to identify, which complicates compression attempts. Searching for the patterns in the overall stream slows down the process as well as increasing the memory requirements. Separating the address streams created by individual instructions exposes simple patterns.

3.3.1 Implementation

PSnAP capture is performed while running the application of interest on an HPC resource. A compiled binary is instrumented to include PSnAP code and the address stream is processed on-the-fly. The result of the instrumented run is a set of PSnAP profiles, one for each core used during execution.

PSnAP capture is implemented as an instrumentation tool and library with a binary rewriter. The binary rewriter, PEBIL, is an open source light-weight binary instrumentation tool that can be used to capture information about the behavior of a running executable. It works on the Linux/X86 instruction set.

The infrastructure provided by PEBIL enables PSnAP to collect all of the requested virtual addresses into a buffer. Each time the buffer reaches capacity, the application execution is paused, register status saved and the buffer is sent to a function within the PSnAP library. The PSnAP library processes the buffer and then returns control to the application execution.

The following section describes our approach to processing the buffer of virtual addresses. We begin with an overview of the representation and then move on to describe the methods of recording, decoding and replaying address streams.

3.3.2 Stream Representation

We achieve over 10,000X compression on address streams primarily through the instruction-specific address stream representation. Three core concepts guide the compression technique.

1. Address streams generated per instruction are simpler than those generated per basic block, loop, or application.
2. Any instruction address stream can be described in terms of a starting address and a stride pattern.
3. Describing an address stream in terms of strides is often, but not always, more succinct than describing it in terms of addresses.

```
for i=0 , limit
  sum += A[i]
  sum += B[1]
```

Figure 3.2: A very simple Loop.

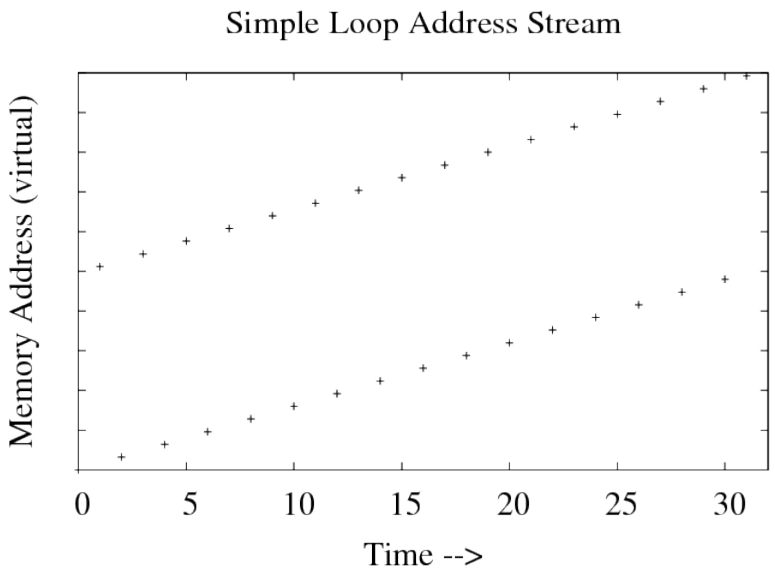


Figure 3.3: A simple address stream.

Before continuing with a discussion of the three guiding concepts, a simplified example is presented in order to facilitate the understanding of complex patterns seen during the discussion. Figure 3.2 presents a very simple loop with two load instructions. Each load instruction operates on a separate area in memory, namely A and B. The address stream generated by this loop can be seen in Figure 3.3. This is a zoomed in view of the address stream showing only the first few addresses. It is apparent that each of the addresses occupies its own space on the x-axis as they are displayed in order, serially. The addresses requested at an offset to the address of A occupy the top half of the figure; those calculated with respect to B occupy the bottom half. This type of representation is used throughout the description of PSnAP.

This section illustrates the guiding concepts of this technique using an example instruction taken from the most dominant block in the NAS benchmark FT. This loop is the dominant loop when running dataset A, compiled for four cores by the Intel compiler on a Nehalem processor. The code block begins at source file ft.f line 1136.

Address streams generated per instruction are simpler than those generated per basic block, loop, or application. This level of granularity was chosen because patterns within small regions of memory are often simple and easily compressed. In almost all cases, instructions act on a single region of memory and display the same level of simplicity [46]. Dynamically discovering the active memory regions is very expensive; but instructions can be discovered statically and therefore, instructions are used to guide compression.

Memory instructions that reside within a regular loop structure and whose memory address is calculated using a combination of loop indexes will display regular patterns. An example demonstrates the benefits of focusing on per instruction address streams. The code shown in Figure 3.4 is from the most dominant loop in FT. Taken as a whole, the address stream that results from this code may seem complex. In fact, the code as supplied contains insufficient information to figure out the pattern. In order for a developer to reason about these access patterns accurately a search of the source code must be performed in order to trace param-

```
1     n1 = n / 2
2     lk = 2 ** (l - 1)
3     li = 2 ** (m - l)
4     lj = 2 * lk
5     ku = li + 1
6
7     do i = 0, li - 1
8       i11 = i * lk + 1
9       i12 = i11 + n1
10      i21 = i * lj + 1
11      i22 = i21 + lk
12      if (is .ge. 1) then
13        u1 = u(ku+i)
14      else
15        u1 = dconjg (u(ku+i))
16      endif
17
18      do k = 0, lk - 1
19        do j = 1, ny
20          x11 = x(j,i11+k)
21          x21 = x(j,i12+k)
22          y(j,i21+k) = x11 + x21
23          y(j,i22+k) = u1 * (x11 - x21)
24        enddo
25      enddo
26    enddo
```

Figure 3.4: An excerpt from the dominant loop of FT.

eters back to values saved in a configuration file. This is a tedious and error prone process.

Breaking down the code block and, therefore, address stream into its constituent pieces makes reasoning about the access patterns significantly easier. Examining specifically the lines between 20 and 32 it becomes apparent that there are actually 2 loads and 2 stores in the address stream. Specifically, two loads from x and two stores into y. Our examples focus specifically on these instructions and use dynamic information to discover the pattern.

Look specifically at line number 20 in Figure 3.4 and specifically at the load from array x. This is Fortran code and so the inner steps are taken using loop

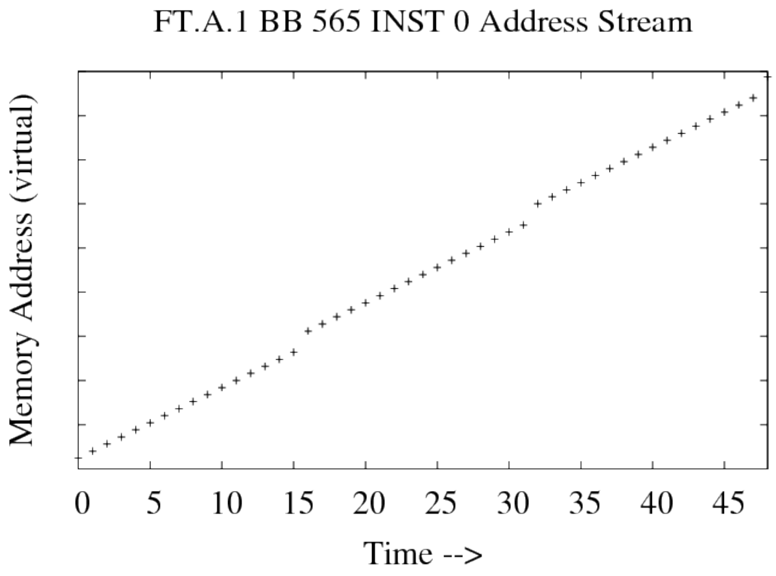


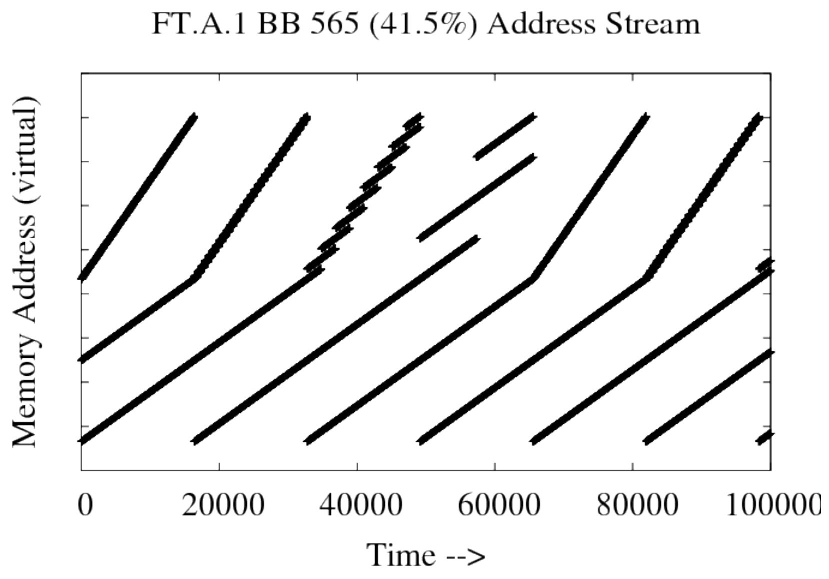
Figure 3.5: The first 48 addresses in the stream for the load from buf

index j. Loop index j is incremented by one ny times, the loop index calculated by $i11 + k$ is incremented $lk - 1$ times. This pattern can be seen in Figure 3.5. The address space is covered by the y-axis. Time is represented along the x-axis (one address per unit time). The graph is created by capturing each virtual address as it is requested. A mark is made on the graph for that address and the time (the x-axis) is incremented by one. Viewing the figure reveals that ny must be initialized to the value 15.

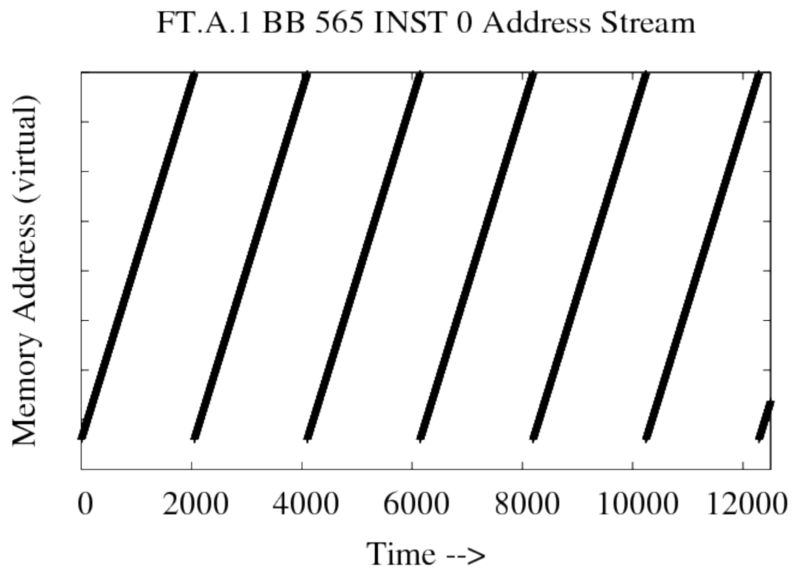
This is only the start of the addresses from a single instruction in the basic block. When the addresses are viewed for all of the instructions over a longer period of time, a more complex pattern is visible.

Figure 3.6(a) shows the address stream for the entire dominant basic block of FT, the code block starting at line 20 in Figure 3.4. The graph is created using the same method as figure 3.5, but this figure shows a larger portion of the address stream, the first 100,000 addresses.

There are two visible patterns in the stream, the upper half has four smaller patterns that are repeated and the lower half is a single consistent pattern. The upper half corresponds to the stores going into y in the example code, and the lower half corresponds to the loads from x. The two address streams are actually



(a) Address Stream: Dominant block in FT.



(b) Address Stream: Single instruction in dominant block.

Figure 3.6: Two views of the address stream from the dominant block of FT.3.4

generated by four pairs of instructions, because the compiler unrolls this loop twice.

Contrast the view of the full address stream to Figure 3.6(b) that shows the address stream for only the first instruction of the block (out of 8). It covers a smaller area (address space) and presents a much more simple pattern. The length x-axis in this figure is one-eighth of the previous one, because we are filtering for addresses from only one of the 8 instructions; the same period of time is covered.

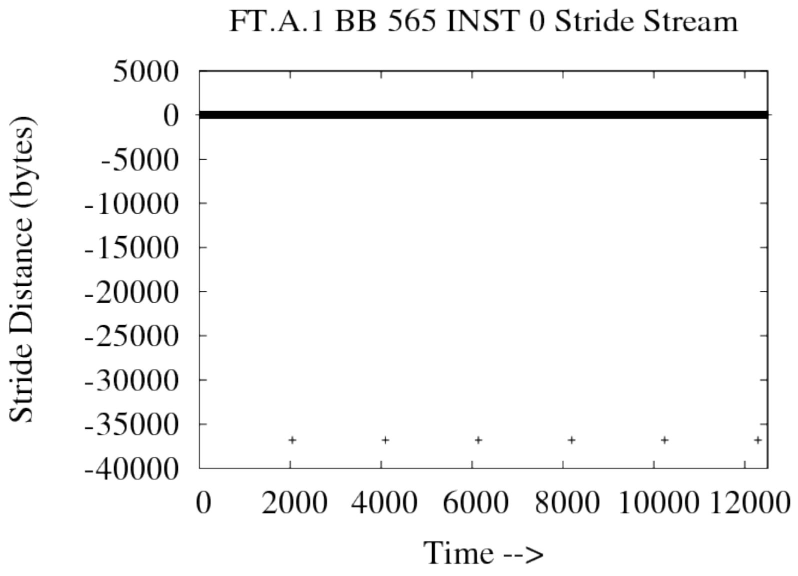
The pattern seen in Figure 3.6(b) can be succinctly represented using strides[2]. The range of addresses produced by this pattern is fairly large, but over the entire execution only four strides are ever encountered. Figure 3.7(a) shows the same range of time, but each point is the distance between the previous and current address, rather than the addresses. In this figure there are 2047 small strides (the thick line across the top) followed by one large backward stride (size 36,816 bytes). This is difficult to see at this scale. Figure 3.7(b) is a zoomed in view of the first 48 strides in this stream; this view reveals that there are 15 strides of size 16 followed by a stride of size 48. We refer to this repeated pattern as the instruction's *stride pattern.* Notice that the stride pattern seen in Figure 3.7(b) matches directly to the address stream seen in Figure 3.5.

Any instruction address stream can be described in terms of a starting address and a stride pattern. Continuing with the current example, a manual inspection of the stride pattern reveals a fixed stride pattern that can be expressed as an expression (very much like a regular expression) where A, B, C, and D are the recorded strides and the exponents indicate the number of times to repeat a pattern.

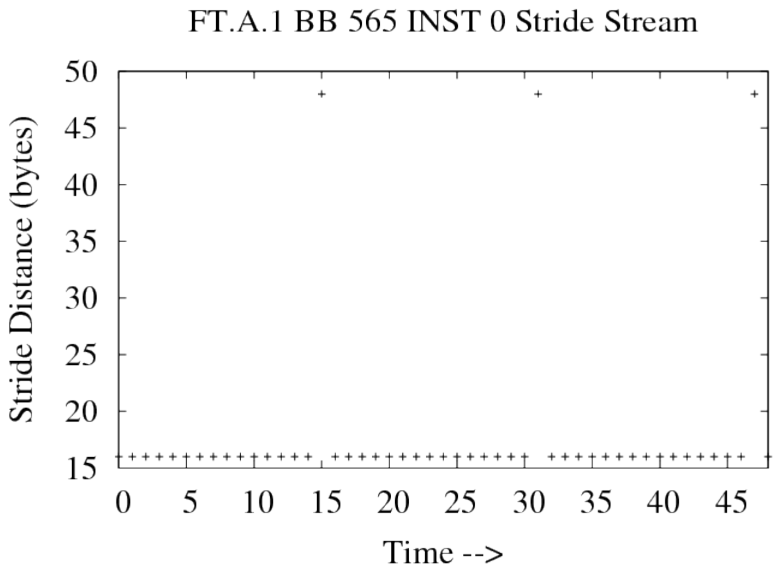
$$\begin{aligned} &((A^{15}B)^{127}A^{15}C)^{16384}A^{15}D \qquad (3.1) \\ &where: A = 16, B = 48, C = -36816, D = -18384 \end{aligned}$$

It should be obvious that any address stream can be described in these terms. Even in a worst case scenario, a completely random stream where no two strides are ever repeated, the stream can be described by simply listing the distances between each address.

[2]A stride is the distance between two addresses in a sequence.



(a) The stride stream.



(b) The stride stream zoomed in to see the first 48 steps in (a), this reveals the pattern that looks like a thick black line in the previous Figure.

Figure 3.7: Reducing an address stream to strides allows for a much more succinct representation of the stream.

Describing an address stream in terms of strides is often more succinct than describing it in terms of addresses. In the FT example it is possible to express the first phase of this address stream using less than 70 bytes. The first phase of this address stream represents approximately 23 Mb of addresses, this is a compression of over 300,000 times. A large portion of address streams are well represented in this manner. In fact, our experiments show that a large portion of instruction-specific address streams are much more simple than this example. We show that out of over 13,000 instructions in the NAS Parallel Benchamrks, only 29 fail to fit this model. Not only is there a high compression rate, but the address stream pattern has been encapsulated in a way that is easily accessible to the reader.

The amount of space required for this representation depends directly upon the number of distinct strides that occur in the address stream and the complexity of the pattern. For instance, in a random address stream, one that may result from a gather operation, there are as many strides as there are addresses. This type of address stream is not a candidate for this representation. It is identified as such (early in instrumentation) and alternate means of recording are employed.

3.3.3 Recording Stride Patterns

PSnAP receives the addresses as a member of a triple. The triple contains the basic block ID, the instruction ID, and the address. The triple is put into the buffer by the instrumentation code written within PEBIL. The first address received for each basic block is recorded in an initialization step. From that point only the strides are recorded.

The stride patterns are recorded by counting occurrences of unique strides. Each stride encountered is searched for in the list of previously encountered strides. When a new stride is encountered it is put at the end of an array list (*strides*) at index i. Another array *count* is updated so that $count(i) = 1$ indicating that $stride(i)$ has been encountered once. In addition, each time a new stride is encountered a snapshot of the count array is recorded. Every repeat of a stride results in a linear search to find the index of the stride in order to increment $count(i)$.

The cost of updating the *strides* and *counts* arrays is not as high as it seems. A linear search is performed to find the index of each stride, but due to the nature of the patterns the vast majority of strides are matched on the first index. In the examples shown in this section the matches that occur at the first index make up 94% of the total accesses. Additionally the number of strides is typically much smaller than the number of addresses. In all of the NPBs the largest number of unique strides for a single instruction is 15.

Table 3.1 shows an example of this data. There are four strides recorded. The first time that a stride of size 48 is encountered, a stride of size 16 has already been encountered 15 times. This can be seen in the first row of the stride history in Table 3.1. At the end of execution the snapshots contain the values needed to create the expression in Eq. 3.2 along the diagonal. The series of snapshots is referred to as a *stride history*.

Table 3.1: Stride history for the first instruction of the dominant block in FT.

Stride	16	48	-36816	-18384
Count	377487360	24903680	131071	131072
Stride History				
1	15	1	0	0
2	1920	127	1	0
3	31458240	2080831	16384	1

This example illustrates the basic idea behind the PSnAP approach. Specifically, the repeated stride pattern can be found by examining a stride history. In this example the diagonal contains the important values for the pattern.

3.4 PSnAP Post-Processing

The post-processing step can be performed on any desktop system, there is no need to use an HPC system. The goal of this step is to characterize each instruction and transform the format of the raw profile in such a way that makes it more easily replayed. This section starts by describing the categories available for characterization (decoding) and then goes on to describe the replay format.

3.4.1 Decoding Stride Patterns

Recall that the recording stage results in a format containing a list of strides, a list of counts, and a stride history. After the stride histories have been recorded they must be decoded in order to prepare for replay. Decoding refers to the process of transforming a stride history into a stride pattern.

Decoding involves three steps. First, each instruction is categorized (*identification*). Second, transformations are performed on the stride histories to reveal the stride patterns (*reduction*). Lastly, the stride patterns are used to guide address stream reconstruction (*pattern*).

Identification involves classifying each stride history. The following pattern classifications are defined:

Constant Refers to the same single memory address each time it is executed.

Simple Repeat Steps through a range using at least two strides. The defining characteristic of this pattern is that each stride is used a number of times that is evenly divisible by the sum of the number of times the strides that come after it are used.

Simple Alternating Is comprised of two or more *simple repeat* that alternate in the address stream.

Complex Repeat Is comprised of a combination of *simple repeat* and *simple alternating* patterns. The patterns are combined using a master pattern.

Undesignated The range of addresses is covered in a random manner or using an index array making the pattern undetectable. This approach would not result in any space savings for this case. Rather than applying it they are identified and handled separately using a probability model.

The *simple repeat*, *simple alternating* and *complex repeat* classifications result from nested loop patterns moving through multi-dimensional data. Simple repeat is the basis for each of them.

Throughout the discussion on decoding the following definitions apply:

A-Z represent the values in the strides array (in order).
c_i is the value in the i^{th} position of the counts array.
$h_{(i,j)}$ is the value in the i^{th} row and j^{th} column of the stride history.

Simple Repeat

A simple repeat pattern results from a nested loop structure stepping through array data. A strided walk through a 1D array is the most basic example. The number of strides collected will depend on the dimension of the data and the depth of the loop structure. Table 3.2 shows the data for a simple repeat pattern taken from the FT benchmark.

Identification. Simple repeat patterns are identified using the counts array (row 2 of Table 3.2. Each count in the line must be evenly divisible by the sum of counts that fall after it.

$$\forall i(c_i \% \Sigma_{j=i+1}^{n} c_j = 0) \tag{3.2}$$

Reduction. Once the instruction has passed the identification test, the next step is to reduce the stride history into representative expressions. The reduction is performed only on the last line of the stride history.

Divide each value by the sum of the values that fall after it in the same row. Given that c_i is the value in position i apply the following.

$$c_i = c_i / \Sigma_{j=i+1}^{n} c_j \tag{3.3}$$

The repeat value is set to the last value in the counts row. The reduced pattern is therefore: (15,255,2047,1) repeated 7 times.

Replay. Replaying the pattern is straight-forward, the pattern is the following.

$$(((A^{15}B)^{255}A^{15}C)^{2047}(A^{15}B)^{255}A^{15}D)^{7} \tag{3.4}$$
$$where: A = 4096, B = -61424, C = 16, D = -134217712$$

It appears to be unnecessary to collect the pattern from the last line of the stride history. From Table 3.2 it is apparent that the same values appear along the diagonal of the stride history. However, the simple repeat pattern is often embedded within simple alternating and complex repeat patterns. In those situations the full diagonal is not available.

Simple Alternating

A simple alternating pattern consists of two or more simple repeat patterns. This can occur during a phase change in the execution or after jumping to a new portion of data that may or may not have the same shape as the previous data. Table 3.3 shows the data for a simple repeat pattern taken from the FT benchmark.

Identification. The simple alternating pattern modulo test will fail for simple repeat. At least two of the counts are alternating; this implies that their total counts will be related based on a factor. We refer to this is as the *alternation factor*. The test for identification involves separating the alternating patterns and testing the remaining patterns separately for simple repeat. An additional test is then performed to ensure that the patterns discovered actually add up to the final counts.

Reduction. In Table 3.3 the modulo test fails at positions 3 and 4 in the counts data. Line 3 of the stride history corresponds to the first occurrence of the stride at position 4. The first step in the reduction is to remove any counts associated with the beginning of the second pattern. Two new data series are created. Let p be the new series and define $p_{i,j}$ where $i = 1|2$ be the value in the i^{th} row and the j^{th} column.

$$p_{1,i} = h_{3,i} \% \Sigma_{j=i+1}^{n} h_{3,j} \tag{3.5}$$

$$p_{2,i} = h_{3,i} - p_{1,i} \tag{3.6}$$

Each of the two new rows are now subject to the identification and reduction performed for simple repeat. The results of this reduction can be seen in the last two lines in Table 3.3.

Identification. The pattern has to now undergo one further test to ensure that it is in fact a simple repeat pattern. Essentially, this is a check to ensure that the final counts can be created by replaying the discovered pattern. Let f_i be the final value in the stride history for row i, $p_{1,i}$ be the value in the i^{th} position of the first pattern, and $p_{2,i}$ be the value in the i^{th} position of the second pattern.

$$f_4 * p_{2,2} + (f_3 + 1) * p_{2,2} = c_2 \tag{3.7}$$

Replay. The alternating factor in this case is determined to be 16384 — the first pattern is done 16384 times before the second pattern is used. It is more difficult to determine how many times the second pattern should be used. This is essentially another pattern in the stream. The current solution is to alternate evenly, improving this policy is the subject of future work.

The stride pattern to replay for this instruction is as follows.

$$\begin{aligned} &(((A^{15}B)^{127})^{16384}A^{15}C \\ &((A^{15}B)^{63})^{16384}A^{15}D)^{131072} \\ where: &A = 16, B = 48, C = -36816, D = -18384 \end{aligned} \tag{3.8}$$

Complex Repeat

The last in this group of patterns is the complex repeat pattern. The complex repeat pattern can comprise both simple repeat and simple alternating patterns. The pattern in the top half of Figure 3.6(a) is a complex repeat pattern taken from FT. This pattern is found in the fourth instruction of the most dominant basic block. Figure 3.8 shows the contribution of the fourth instruction to the basic block pattern. Its stride history is presented in Table 3.4[3]. Four distinct patterns can be seen in the first 8000 addresses of the figure. The replay pattern reflects these patterns.

Identification. The complex repeat pattern is more involved to identify than the previous two examples. The reduction steps must be complete and then each of the resulting patterns are tested for simple repeat and simple alternating.

Reduction. The complex repeat pattern is a combination of other patterns, therefore, the first reduction step is to identify the individual patterns. The *dominant lines* have to be identified. A dominant line is one that describes a complete

[3]The pattern created by this basic block has two phases (a simple repeat) the second phase has been removed in order to make the example more readable.

Table 3.2: The stride history for a simple repeat.

Simple Repeat				
Raw Profile (Stride, Counts, Stride History)				
stride	4096	-61424	16	-134217712
count	62914560	4177920	16376	7
Stride History				
1	15	1	0	0
2	3840	255	1	0
3	7864320	522240	2047	1
Replay Pattern				
pattern	15	255	2047	1

Table 3.3: The stride history for a simple alternate.

Replay Pattern				
Raw Profile (Stride, Counts, Stride History)				
stride	16	48	-36816	-18384
count	377487360	24903680	131071	131072
Stride History				
1	15	1	0	0
2	1920	127	1	0
3	31458240	2080831	16384	1
Replay Pattern				
pattern	15	127	1	0
pattern	15	63	0	1

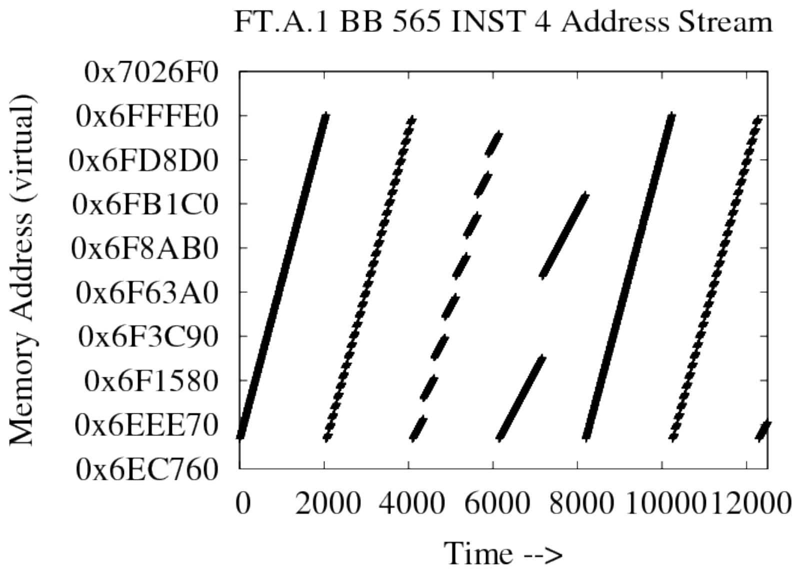


Figure 3.8: The address stream for the fourth instruction of the most dominant block in FT.

pattern. Both simple repeat and simple alternating patterns have dominant lines as well, but they are trivial to locate because they are always the last line in the stride history. A dominant line is always followed by a line with a 1 in the diagonal.

Definition 1. $history(i)$ *is dominant* $\iff history(i+1, i+1) = 1$

Definition 2. $final(i) == j \iff dom(i,j) = 0 \ \& \ \forall_{k=j+1}^{n} dom(i,k) = 0$

Each dominant line is further reduced.

1. Reduce each dominant line by the values in the dominant line directly above it.
$$dom(i,j) = dom(i,j) - dom(i-1,j) \tag{3.9}$$

2. Subtract one from the position in each dominant line that corresponds to the final value in the dominant line above it.
$$dom(i, final(i-1)) = dom(i, final(i-1)) - 1 \tag{3.10}$$

The last dominant line is not a simple repeat or simple alternate pattern. This line is referred to as a *master line*. A master line guides the repetitions of the patterns that come before it. This line is reduced to only the entries that correspond with $final$ values in the patterns above it.

$$master(j) = 0 \iff \nexists i | final(i) = j \tag{3.11}$$

Each of the dominant lines is passed through the reduction performed for simple repeat or simple alternating. In this example all of the lines are simple repeat. The reduced dominant lines and final pattern are listed in Table. 3.4.

Pattern. The replay for this pattern is performed in the same way that each of the previous patterns. The only addition is that the patterns are grouped together and repeated 4095 times.

$$\begin{aligned}
(A^{15}B)^{127}A^{15}C \\
((A^{15}D)^{3}A^{15}E)^{31}(A^{15}D)^{3}A^{15}F \\
((A^{15}D)^{15}(A^{15})G)^{7}((A^{15}D)^{15})^{7}A^{15}H \\
((A^{15}D)^{63}A^{15}I) \\
((A^{15}D)^{63}A^{15}J) \\
where: A = 16, B = 336, C = -73392, D = 48 \\
E = 1200, F = -72528, G = 4656, H = -69072 \\
I = 18480, J = -55248
\end{aligned} \tag{3.12}$$

A representation such as the above is detailed yet compact and is much more human readable than a raw address stream. If needed a human or a post-analysis tool can tell a lot about an application or a section of an application from examining patterns such as the above. We also provide the ability to ”zoom in” on specific loops or functions and show their patterns. PSNaP’s ease of use allows one to recapture the same stride patterns after a change. For example, a recompile with new flags or a code restructuring, to examine what has changed or improved.

Table 3.4: The stride history for a complex repeat.

Complex Repeat									
Raw Profile (Stride, Counts, Stride History)									
Stride	16	336	-73392	48	1200	-72528	4656	-69072	18480
Count	3.7E8	6.2E6	32768	1.6E7	1.5E6	32768	3.2E5	32768	32775
Stride History									
1	15	1	0	0	0	0	0	0	0
2	1920	127	1	0	0	0	0	0	0
3	1935	127	1	1	0	0	0	0	0
4	1980	127	1	3	1	0	0	0	0
5	3840	127	1	96	31	1	0	0	0
6	4080	127	1	111	31	1	1	0	0
7	5760	127	1	216	31	1	7	1	0
8	6720	127	1	279	31	1	7	1	1
9	7680	127	1	342	31	1	7	1	1
10	3.1E7	5.2E5	4096	1.4E6	1.2E5	4096	28672	4096	4096
Reduced Dominant Lines									
1	1920	127	1	0	0	0	0	0	0
2	1920	0	0	96	31	1	0	0	0
3	1920	0	0	120	0	0	7	1	0
4	960	0	0	63	0	0	0	0	1
5	960	0	0	63	0	0	0	0	0
6	0	0	4095	0	0	4095	0	4095	4095
Replay Pattern									
1	15	127	1	0	0	0	0	0	0
2	15	0	0	3	31	1	0	0	0
3	15	0	0	15	0	0	7	1	0
4	15	0	0	63	0	0	0	0	1
5	15	0	0	63	0	0	0	0	0
6	0	0	4095	0	0	4095	0	4095	4095

3.5 PSnAP Replay

The goal of PSnAP replay is to create a synthetic address stream as fast as possible. The algorithm is designed to replay the synthetic stream while using as few instructions as possible. For this reason a representation of the patterns has been developed that covers all of the categories described above.

The representation is essentially a set of equal length global arrays accompanied by a control flow data structure. The control flow data structure guides the replay to each instruction. The instruction is represented to the control flow structure as a starting address and an index into the global arrays.

Replay Pattern					
First Addr: Ox6DC1C0					
Index	Stride	Count	<	==	State
1	16	15	1	2	
2	48	127	1	3	
3	16	15	3	4	
4	-36816	16384	1	5	
5	16	15	5	6	
6	48	127	5	7	
7	16	15	7	8	
8	-18384	16384	5	1	

Table 3.5: The PSnAP Replay profile for the first instruction of the most dominant block in FT.

Table 3.5 shows the replay pattern as it is represented in memory during replay for the first instruction of the dominant basic block in FT. This instruction's pattern is also used as the example of a simple repeat pattern.

There are four global arrays: stride, count, less than (<), and equal (==). In this abbreviated example the index for this instruction is one. The arrays are global and hold all of the information for each of the instructions and so this pattern will be followed by another pattern for the next executed instruction.

```
address = prev_address + stride[idx]
if( state[idx] < count[idx] )
  state[idx]++
  idx = lt[idx]
else
  state[idx] = 0
  idx = eq[idx]
```

Figure 3.9:

Psuedo-code describing PSnaP's replay algorithm.

The algorithm is written out using pseudo-code in Figure 3.9. The state array is used to keep track of the current progress. When the control flow structure chooses an instruction, the index is looked up in the state array and compared to the value in the count array. If the value in the state array is less than the value of in the count array the value in the state array is incremented and the index is replaced by the value in the less than array. If the value in the state array is equal to the value in the count array, the value in the state array is set to zero and the index is replaced by the value in the *equal* array.

All of the categories described in the previous section can be represented in this manner. The choice of address then requires two loads, one comparison and two integer operations. This is expected to be slower than the execution of the application primarily because of the loads. In addition to the PSnAP replay, the control flow replay must be performed as well. This takes additional time as well.

3.6 Control Flow Compression

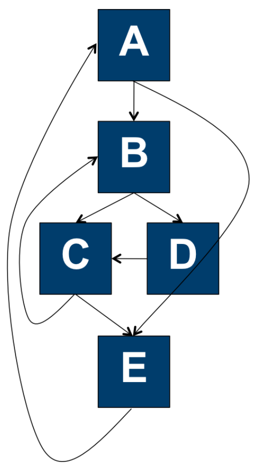
As described thus far, PSnAP provides a mechanism to record and replay address streams for individual instructions. Full address stream replay requires that the instruction streams be replayed together in the correct order. The order is preserved using control flow compression. The control flow compression used by PSnAP follows the method used in Path Grammar Guided Trace Compression (PGGTC) [21] very closely. The following is a brief overview of the approach.

The control flow graph of an application is a directed graph that describes the path taken through the code. The application code is broken down into basic blocks, a set of instructions with a single entry and single exit, which are represented as nodes in the graph.

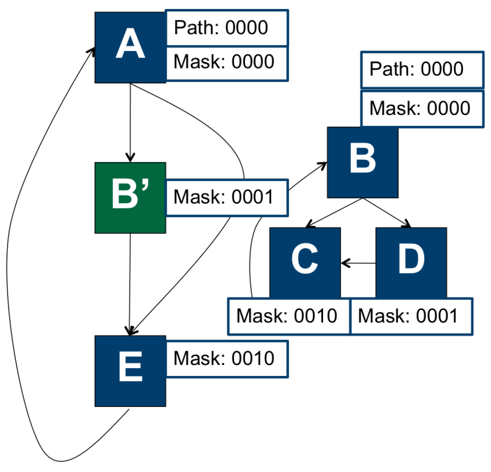
There are two levels of constructs in the representation, nodes and aggregates. The nodes are the basic blocks and the aggregates are loops or function calls. An aggregate construct is different from a node only because it contains either other aggregate constructs or nodes. The representation is loop-centric.

Loop structures and basic blocks are identified statically using a binary rewriting tool PEBIL. The loops and basic blocks are each assigned a unique id. The loop ids are unique over the entire code and the block ids are referred to as masks and are unique only within the containing loop.

Each loop in the application is treated separately. Figure 3.10(a) shows a simple example of a control flow graph. This graph contains a loop with a nested loop. Figure3.10(b) demonstrates how the nested loop is separated. the head basic block of the nested loop is promoted to be an aggregate construct. The promoted node is always an entry node to the nested structure.



(a) The original loop.



(b) The resulting loop.

Figure 3.10: Loop separation in the control flow graph.

Instrumentation code is inserted at the loop heads and at the beginning of each basic block. This code is responsible for recording the path taken through the loop and the iteration count for each entry. Table 3.6 shows the data recorded during the instrumentation run, specifically the path history and the iteration history.

The path history describes the paths taken through the loop. Individual paths taken through a loop are expressed in a bit map. The bit map is maintained by the loop head and updated by each visited basic block. Each basic block, or dummy basic block that represents a nested loop, is statically assigned a mask. Instrumentation code at each basic block performs a bitwise-OR with the loop bit map. At the end of a single iteration of the loop each flipped bit in the map represents a basic block taken. The data in Table 3.6 indicates that blocks one and three are taken. Block 0 is always taken as it is the head node of the loop.

The ordering of the masks for basic blocks is key for accurate replay. The 1 bits in the bit mask indicate which basic blocks where executed and the order they were executed in. If the basic blocks in figure 3.10(a) were labeled as B=00, C=01, and D=10 (a valid breadth first ordering) the execution of C and D would be swapped during replay. A topological ordering [59] is used to ensure that the path represents a valid execution ordering of basic blocks.

Each sub-loop requires only a single bit in the parent loops bit map. Using a topological ordering within a loop always assigns a mask of 0 to the head block of a loop. This block must always be taken and the mask is therefore, not needed. The space is used instead to hold the mask that the sub-loop would have been assigned if it had been just a normal basic block. This means that the head block of a sub-loop is able to play the role of the dummy basic block in the parent loop.

At the end of each loop iteration the path taken through the loop is compared to the path taken on the previous iteration. If the paths match, a counter is incremented, if not, the new path is pushed on a stack of paths, called the *path history*. A limited number of paths are recorded to save space. In the example in Table. 3.6 the same path is taken each time through the loop.

If the number of allowed paths is exceeded, the loop is labeled as having overflowed. A snapshot of the current path history is taken and saved. At that point the history converts to only counting the paths taken, the order is no longer

Table 3.6: The control flow representation for the most dominant loop in FT.

	Loop 7				**Loop 8**
ID	563	564	565*	566	565
Mask	0	1	2	3	0

Path History			
Path	**Count**	**Path**	**Count**
0000 0101	4798283776	0000	4798283776

Iteration History			
Iter.	**Entries**	**Iter.**	**Entries**
16	256	1	4798283776
64	64		
256	16		
1024	4		
4096	1		

maintained. The number of entries maintained in the history is configurable, for the experiments using the NPBs 20 entries was adequate.

A simple iteration history is kept for each loop to keep track of iteration counts for each entry. Each time the loop is entered externally a new counter is initialized. The count is incremented for each iteration, and upon exit the count is pushed onto a stack. If a loop is iterated over the same number of times by successive external entries a counter is incremented rather than saving multiple copies of the same value.

There are several differences between the PSnAP implementation and the original Path Guided Control flow Compression Technique (PGCCT) implementation. PGCCT is non-lossy whereas PSnAP is lossy. Specifically, rather than collecting new paths indefinitely, PSnAP allows for a finite number of paths to be collected before simply counting unique paths rather than keeping track of the order they appeared in. For example, if two paths were taken through a loop in an alternating manner 1 million times, PGCCT would have 1 million entries in the stack of paths. PSnAP would have only 2, each with an associated count of

500,000. This saves a large amount of space, but results in a potential loss of accuracy, if the pattern cannot be determined. As a hint to the pattern a snapshot of the path history at the point of overflow is saved in the profile.

The other major difference between the implementations is that the bit maps in PGGTC were limited to 32 bits. If the path through a loop contained more than 32 basic blocks a hash map was used. The bit maps in PSnAP have been implemented to expand a byte at a time indefinitely.

The result is a highly compressed trace, even of long running or complex applications or benchmarks, that can be easily shared, even by email, to convey the memory behavior of applications.

3.7 Evaluation

The accuracy and performance of PSnAP were evaluated through a series of experiments. The experiments include a coverage survey, manual inspections of key basic block address streams, cache hit rate comparisons, and a performance evaluation.

The experiments were conducted using the NAS Parallel Benchmarks run on four cores. All of the experiments were run on Dash[4]. Dash is a 64 node system housed at SDSC. Each node is equipped with two quad-core 2.4 GHz Intel Nehalem processors and 48 Gb of memory.

A coverage survey was performed in order to verify that an adequate percentage of instructions qualify to be represented using stride patterns. The accuracy of the synthetic address streams was evaluated in two ways. First, the instruction-specific streams were directly compared at a basic block level. This ignores the interactions between instructions from the comparison. However, the control flow is included in this comparison, because the path history and iteration history are both required to create the replay for this test.

Second, the cache miss rates of the synthetic address streams were compared to those of the observed address stream. The synthetic address streams prove to

[4]`http://www.sdsc.edu/us/resources/dash/dash_system_access.html`

be very accurate; the instruction-specific streams are non-lossy with a 100% match rate. Cache miss rates were reproduced with an average error of 0.8% (i.e less than 1%) in L1.

The performance evaluation of PSnAP included examining the achieved compression rates and the overhead of generating the profiles and synthetic streams. The compression rates are competitive with the state of the art, often surpassing it. At the same time the slowdown is significantly less; PSnAP incurs an average slowdown of 90X. This slowdown is achieved with no sampling and using an unoptimized version of the instrumentation code. The slowdown is already very low for such an instrumentation approach and we believe that the slowdown can be further reduced with the addition of sampling and optimizations.

3.7.1 Coverage

Before examining the accuracy of PSnAP for the benchmark set as a whole, it is important to ensure that individual instructions are being properly represented. This cannot be done unless a significant portion of instructions are represented by the defined stride patterns from section 3.3.3. A coverage test was conducted to measure the number of instructions in the benchmarks that qualified as candidates. Of the over 30,000 instructions evaluated in the NPBs, only 29 were not represented using the defined patterns.

Table 3.7: Instruction-level pattern coverage statistics for NPBs.

Category	**BT**	**CG**	**EP**	**FT**	**IS**	**LU**	**MG**	**SP**
constant	219	306	27	92	57	1177	305	2153
simple repeat	2062	0	3	16	5	1810	19	781
cmplx. repeat	995	0	0	119	0	0	125	1762
simple alt.	178	0	2	4	0	0	304	566
undesignated	0	2	0	0	11	0	16	0
Total	3455	386	32	231	73	2987	769	5262

Table 3.7 shows the coverage statistics for the categories in PSnAP. A majority of the instructions, 61%, are represented with simple repeat, complex repeat

and simple alternating. The majority of the remaining are constant, 38% of the instructions are constant. This is encouraging for performance, as the constant pattern requires almost no processing and no memory to replay.

The undesignated instructions, which occur in CG, IS and MG, were manually examined. Each of them results from the use of index arrays and are in fact not candidates for this representation. For these cases, a random access stream is generated during replay.

3.7.2 Pattern Resolution

The reproduction of fine-grained patterns has historically been a weakness in synthetic stream generation. A high level of resolution is very important when studying memory behavior, especially when prefetching is a consideration. Fourier transforms are a common task performed in HPC applications, and they produce complex address stream patterns. Figure 3.6(a) shows such an address stream, which is the dominant pattern in FT. PSnAP is able to reproduce this exact address stream. In order to demonstrate the level of resolution achieved by PSnAP we manually compared the first 100,000 addresses for the dominant basic block in each benchmark. An exact match was achieved for each of the NPBs with qualifying instructions, even the complex pattern shown in Figure 3.6(a).

An exact match with fine-grained patterns does not imply perfect accuracy. There is lossiness within the control flow compression and therefore cache simulation results were used to test the overall accuracy of the synthetic streams.

3.7.3 Cache Simulation Results

Cache simulation comparisons allow for a high-level view of PSnAP's accuracy. We have shown that the fine-grained patterns are reproduced on a per instruction, and per basic block basis, but the lossy nature of the control flow compression leaves an opening for differences between the observed and synthesized streams. We compare the overall cache miss rates for the entire execution as well as perform a more fine-grained basic block comparison.

Table 3.8: Cache structures used for cache hit rate accuracy verification.

ID	**L1** **Sz/Ln/Assoc.** **(KB)/(Bytes)/**	**L2** **Sz/Ln/Assoc.** **(KB)/(Bytes)/**	**L3** **Sz/Ln/Assoc.** **(KB)/(Bytes)/**	**Arch.**
1	32/128/2	1024/128/8		PowerPC
2	256/128/8	9216/128/12		IT2
3	64/64/2	512/64/16		MIPS
4	32/64/8	3072/64/24		
5	32/32/4	128/64/2		Opteron
6	64/64/2	512/64/16	1024/64/48	Budapest
7	32/64/8	256/64/8	2048/64/16	Nehalem
8	64/128/8	4096/128/8	16348/128/16	IBM P6
9	64/64/2	512/64/8		
10	64/64/2	512/64/32		
11	64/64/2	512/32/16		
12	64/64/2	512/128/16		

The cache structures used to examine the accuracy of the streams are described in Table 3.8. This evaluation uses a set of seven memory hierarchies taken from recent and historical HPC systems. Included in the set are PowerPC, IT2, MIPS, Opteron, Budapest, Nehalem, and IBM Power6. The MIPS structure is altered to create four hypothetical structures. The line size and associativity are varied in turn. The structures were chosen to represent small, medium and large caches, with a variety of line size. Some of the structures are two level caches and some are three.

The observed address stream of each benchmark was fed into a series of cache simulators [62, 35]. The cache simulations produce cache miss rates for each basic block in the application and for the entire execution. These cache miss rates are then compared with the cache miss rates that result from the simulation driven by the synthetic streams.

Table 3.9 shows the difference in the miss rates (between synthetic and observed) for the entire execution of each benchmark on the Nehalem cache structure

Table 3.9: Performance of PSnAP on NPBs (4 processors).

Benchmark	**Full Trace Size**	**Compressed Files**		**% Abs Err**		
		PSnAP		**PSnAP**		
	(GB)	**ratio**	**size KB**	**(L1)**	**(L2)**	**(L3)**
BT.A	1,120	84,856X	13,840	0.2	0.1	0.1
CG.A	18	83,020X	232	0.4	0.4	0.1
EP.A	51	1,973,790X	27	0.0	0.0	0.0
FT.A	64	97,203X	690	0.2	0.5	0.4
IS.A	43	134,019X	338	1.3	0.2	0.1
LU.A	599	79,399X	7,908	1.9	1.5	0.6
MG.A	40	19,760X	2,118	1.1	0.7	0.3
SP.A	508	33,359X	15,968	1.4	1.0	0.7

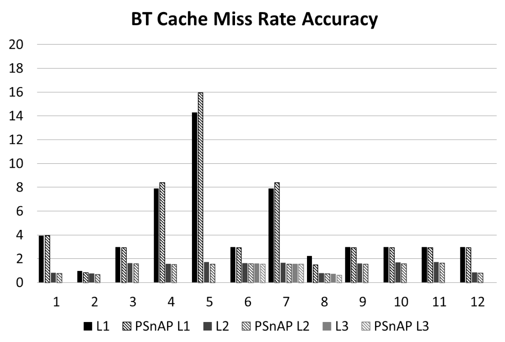
(number 7 in Table. 3.8). PSnAP's overall accuracy is very high. The average error for L1 is 0.8%.

A more detailed comparison of basic block cache miss rates confirms PSnAP's accuracy. For this comparison the cache miss rates across all cache structures were compared for the dominant basic block in each benchmark. Table 3.10 shows the observed and synthetic cache miss rates for the dominant basic block of each benchmark using the Nehalem cache structure. Not surprisingly, IS is the worst performing. The dominant instruction in IS performs a load calculated using an index array. It is categorized as undesignated and a random function is used to generate the synthetic stream. All of the basic blocks containing instructions with recorded stride patterns performed well, the maximum difference is 0.71%.

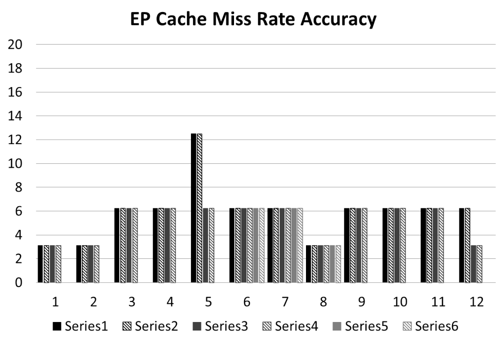
Figures 3.11 a-f show the variation in accuracy for L1 miss rates over all of the cache structures. EP, LU and SP each had perfect or near perfect synthetic streams resulting from the profiles, this can be seen in the in the cache hit rates as well; each has very consistent high accuracy. FT has perfect accuracy in L1, but the L2 and L3 cache miss rates have some error. This error as well as the errors in the other benchmarks is due to the lossy nature of the control flow compression. Several basic blocks in FT interact and the control flow plays a key role in that coordination.

Table 3.10: Accuracy of PSnAP on NPBs at the Basic Block Level for Nehalem Cache (4 processors).

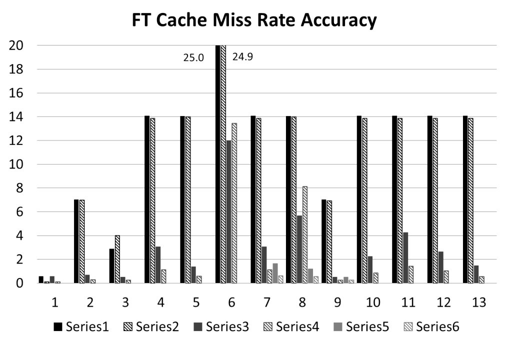
Benchmark	**level**	**Cache Miss Rate**		
		Observed	**Synthetic**	**Abs. Diff**
BT.A	L1	5.29	5.14	0.15
	L2	1.59	1.58	0.01
	L3	1.51	1.41	0.10
CG.A	L1	24.57	23.86	0.71
	L2	7.21	7.32	-0.11
	L3	6.83	6.84	-0.01
EP.A	L1	6.25	6.25	0.00
	L2	6.25	6.25	0.00
	L3	6.25	6.25	0.00
FT.A	L1	13.91	13.52	0.39
	L2	0.93	0.00	0.93
	L3	0.01	0.00	0.01
IS.A	L1	3.67	6.78	-3.05
	L2	1.05	6.45	3.05
	L3	0.78	5.74	3.05
LU.A	L1	6.23	6.27	-0.04
	L2	4.59	4.69	-0.01
	L3	0.71	0.35	0.36
MG.A	L1	11.49	11.01	0.48
	L2	4.21	4.28	-0.08
	L3	3.39	2.13	1.26
SP.A	L1	2.30	2.30	0.00
	L2	2.30	2.30	0.00
	L3	2.30	2.30	0.00



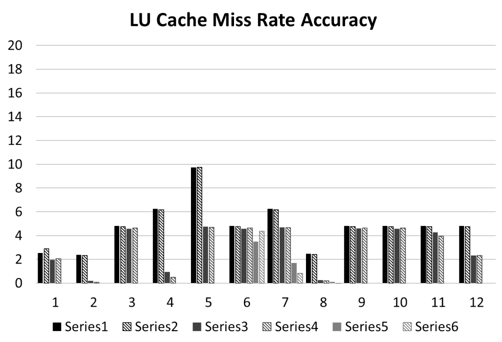
(a) BT



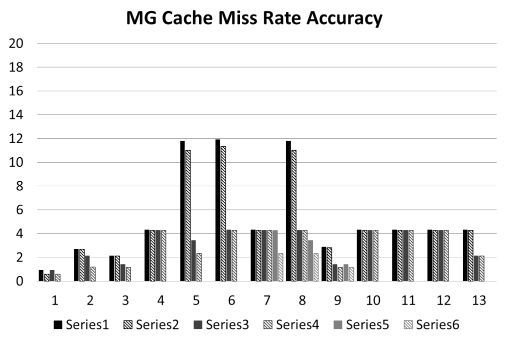
(b) EP



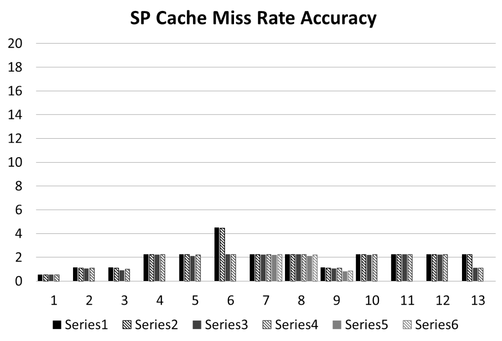
(c) FT



(d) LU



(e) MG



(f) SP

Figure 3.11: PSnAP's Accuracy with respect to Cache Miss Rates in the Dominant Blocks of Each Benchmark

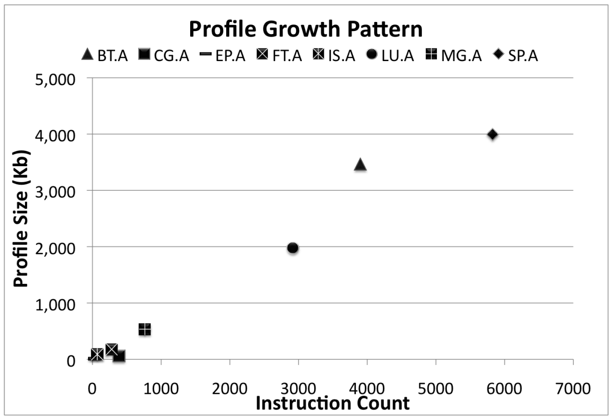


Figure 3.12: Profile size grows as a function dominated by number of instructions.

Our experimental results demonstrate very clearly that the synthetic streams are very similar to the observed in terms of performance. The error is consistently below 1%, with the previously noted exception of IS.

3.7.4 Size and Slowdown

The size and scaling behavior of the memory profiles are major advantages of the PSnAP approach. Each of the benchmarks used for the accuracy evaluation produced memory profiles of less than 2MB, which is easily shared among collaborators. Table 3.9 shows the compression ratios for PSnAP and PGGTC. PSnAP and PGGTC achieve fairly similar compression rates, but PSnAP is more consistent in obtaining high ratios.

The size of each PSnAP profile is a function of address stream complexity rather than running time, or dataset size. Figure 3.12 shows the PSnAP profile size for each of the NPBs plotted against the memory instruction count (determined statically). The size of the profile is a function of loop count, basic block count and instruction count, but is largely dominated by the memory instruction count.

The real advantage of PSnAP over PGGTC is the overhead. Table 3.11 shows that the collection overhead is on average 90X. This is very small for a profiling technique with no sampling. The addition of sampling is planned as future work, leaving the opportunity for even more improvements. We anticipate the addition of sampling to reduce the overhead to 10X.

Table 3.11: Running times for instrumentation and replay.

Benchmark	**Running Times(slowdown)**		
	Exec.	**PSnAP**	**Replay**
	sec.	**sec.**	
BT.A	27	1856(69X)	1170
CG.A	0.7	56(77X)	42
EP.A	3	163(59X)	18
FT.A	2	197(116X)	235
IS.A	0.6	62(112X)	44
LU.A	19	1610(86X)	967
MG.A	0.8	88(117X)	109
SP.A	23	1441(62X)	1304

The replay time is a significant metric because slow replay leads to slow simulations. The current state of the art for replay speeds is to replay at disk read speed (optimistically 250 MB/s). An optimized version of the PSnAP replay tool approaches this rate for some profiles. However, we have devised an alternative replay process in order to guarantee consistent replay speeds at disk speed.

The alternate replay process involves building a temporary (and much larger) representation of the stream on disk. The temporary representation will be too large to move around or inspect directly, but allows for fast replay. Mache [50], is chosen as the alternative address compression scheme. It results in larger representation, the best compression ratio possible is 8X, but is capable of replay speeds at the same rate as disk reads.

We have demonstrate that the PSnAP profiles have a high degree of accuracy, low instrumentation costs, and that is is possible to replay the profiles at disk speeds. The result is a highly accurate, highly compressed trace that is easy to capture and replay whether to drive simulation or other analysis.

3.8 Discussion

Despite the general usefulness of trace-driven memory simulation the collection, handling and storage of memory address traces remains problematic. PSnAP addresses the challenges associated address trace collection including space costs, time costs, accessibility, and proxy inaccuracies by decomposing address streams into smaller simple streams. The repeated patterns in the smaller streams are recorded during dynamic execution using a simple stride history scheme.

Space Costs. PSnAP achieves consistent compression ratios higher than any previously compression technique. Not only are the compression ratios high, but the growth of the profile sizes does not grow in relation to execution time. Once each instruction level stream description has been established the profile stops growing regardless of the number of times that it is repeated.

Time Costs. The compression ratios are achieve while maintaining low execution overhead (on average 90X). We have presented a method for replay that matches the disk speed available. It is also possible to perform direct replay of the profiles, which outperforms disk reads for some benchmarks. Another advantage of PSnAP is that specific areas of interest in the application can be captured in the profiles rather than the entire application. This is desirable for extremely long running applications where any slowdown is a difficult challenge.

Accessibility. PSnAP profiles are human readable and manipulatable. This opens up several new usage scenarios for address streams. Due to their size PSnAP profile are easily shared between collaborators. It is also possible to replay only specific portions of the stream. Specific loops can be identified and played along with the loops nested within them.

Replay check-pointing is also possible. This means that the replay can be performed in sections. It is possible to request only a million addresses at a time. This level of control makes exploring and experimenting with the streams much easier than with other compression methods.

Proxy Inaccuracies. It is not necessary to evaluate benchmarks or application kernels as proxies to real application. The applications themselves can be represented with PSnAP profiles.

PSnAP represents a real advance in the handling of address streams. We note improvements in the areas of space and time costs as well as making the information captured in memory address streams easily accessible to users.

3.8.1 Future Work

We are actively moving forward with improvements and additions to PSnAP. Our goals are to shrink the collection overhead, improve the handling of undesignated instructions, develop tools to ease the exploration of compressed traces, and optimize the replay process.

PSnAP's collection overhead is already manageable, but there are a few optimizations that are possible to shrink it even further. The current implementation collects control flow information and instruction address stream information at the same time, using a shared library. A faster instrumentation approach can be used if the two are separated. On their own the control flow compression and instruction level address stream compression will require fewer operations, simpler logic, and less memory.

It is possible to trade some level of accuracy for instrumentation speed. Many simulations that consume memory address streams implement sampling. The traces can be collected in a way that compliments the consumer's sampling scheme. For instance, if the consumer is only going to observe the first 10% of addresses there is no need to detect patterns for the entire execution.

In addition to course grained sampling it is possible to turn off instrumentation in sections of the application that have "settled" or achieved a stable pattern. This behavior can be detected by the absence of new strides. Leaving the first instruction of the region instrumented in order to check for a change in pattern ensures accuracy, while realizing a significant performance improvement.

PSnAP currently has a single category of undesignated instructions. A lot of previous work [5, 60, 27, 66] has been done that can improve the representation of these instruction streams. One of our goals is to detect different categories of "random" streams and reproduce them with greater verisimilitude.

The exploration of address streams is greatly eased with the use of PSnAP.

As of now individual loop structures can be replayed in isolation and it is possible to replay streams in manageable sized chunks. We plan to improve upon these capabilities and make an interface available for zooming in and replaying specific regions or basic blocks as well.

The replay process is in the process of being optimized. It is our belief that the replay has the potential to be significantly faster than disk speeds. Methods of parallelization using threads are being explored.

3.8.2 Usage Scenarios

The PSnAP format is human readable and reveals a great deal of information about the underlying structure of the application. Evaluating the profiles reveals some of the compiler optimizations performed by the compiler (such as loop unrolling) as well as access patterns, loop depth, and path frequencies just to name a few. The availability of this information has prompted the beginning of two new research projects based on PSnAP profiles, PSnAP guided auto-tuning and PSnAP guided scaling analysis.

The PSnAP profiles can be manipulated to reflect the effects that different compiler optimizations would have on the address stream. By replaying the relevant portions of manipulated PSnAP profiles it will be possible to compare the effectiveness of different combinations of optimizations.

By collecting a series of PSnAP profiles for a single application run across varying numbers of cores one can inspect the profiles and detect patterns of change. Using regression techniques it should be possible to extrapolate the values from the series of profiles in order to create a new profile at a very large core count. It would then be possible to generate a synthetic address stream at a core count that is not currently available or to do scalability studies for scientific applications.

3.9 Conclusion

PSnAP is a lossy address stream compression tool. It accurately represents the address streams for HPC applications and benchmarks using very little space. The profiles allow for a synthetic stream to be generated, manipulated, and measured.

PSnAP combines previous work done on dynamic control flow graph compression with a new technique for compressing instruction-level address streams. The development of the fine-grained instruction-level address stream compression allows for much faster instrumentation as well as the very small profiles. We demonstrate that fine-grained patterns are reproduced address for address and cache hit rates are reproduced with an average error of 0.8%.

Acknowledgments

Chapter 3 contains material reprinted from the proceedings of the Workshops on Languages and Compilers for Parallel Computing. Olschanowsky, C.M. and Tikir, M.M. and Carrington, L. and Snavely, A. *PSnAP: Accurate Synthetic Address Streams Through Memory Profiles.* 2009. [46]. The dissertation author was the primary investigator and author for this paper.

Chapter 3 contains material reprinted from a submission to the IEEE International Symposium on Workload Characterization. Olschanowsky, C.M. and Carrington, L. and Snavely, A. *A Tool for Characterizing and Succinctly Representing the Data Access Patterns of Applications.* 2011. [43]. The dissertation author was the primary investigator and author for this paper.

Chapter 4

Address Stream Extrapolation

4.1 Introduction

The number of processing cores available to high Performance computing (HPC) applications is growing to extremely large scales. New National Science Foundation compute resources located at the University of Illinois at Urbana-Champaign, the University of Tennessee, and the Texas Advanced Computing Center are coming online now and over the next several years; the largest, Blue Waters, the petascale resource at UIUC, will provide 500 times more computing power than today's supercomputers using more than 100,000 processors. Systems of similar size are being deployed by The Department of Energy and NASA. This growth pushes application developers to write code that remains efficient when run across many cores. Understanding the scaling behavior of applications and being able to predict how well an application will run on processor counts not yet available is key to achieving effective utilization of these and other high performance resources.

The scaling behavior of the on-node computation of applications is of particular interest. This is due to the correlation between data-movement (a large portion of the computation component) and both power consumption [44] and performance [55]. Data-movement and power consumption are considered two of the largest hurdles to achieving exascale performance by the Darpa Exascale report [18], this highlights the importance of developing methods to understand and optimize on-node scaling behavior.

The direct extrapolation of high-level memory performance metrics is challenging. It has long been understood that extrapolating the observed runtimes of a parallel application executed on different processor counts is problematic as a prediction technique. The runtime of an application is a high-level metric which hides the true complexity of execution; there are unexpected “cliffs” and turning points in scalability that are due to the complexity of algorithm and architecture interaction. Another high-level metric that does not lend itself to extrapolation is cache miss rate [56]. Even measured at a per basic block granularity, there are drops in miss rates that are difficult if not impossible to predict, without detailed knowledge about the interaction of the address stream and the cache structure.

This work hinges on the hypothesis that while high-level metrics such as execution time and cache hit rate do not extrapolate accurately, there are low-level execution metrics which do, metrics that can be derived from the application’s address stream. These low-level metrics can then be used to derive the high-level metrics.

The data-movement of an application is described by an address stream and is in turn used to re-derive the high level metrics. An address stream is the list of virtual addresses requested by an application during execution. The stream can be used to explore the memory behavior of an application and used to drive simulation studies.

The streams resulting from executions at extremely high core counts are unavailable for at least two reasons, the streams are too large to be collected and/or the resources are not yet deployed at scale. We present an extrapolation method that, when combined with application knowledge, produces synthetic address streams that closely resemble the observed address streams at higher core counts. This approach eases the exploration of application scaling performance and power consumption in advance of resource deployment.

The contributions of this work include the identification of low-level metrics that lend well to extrapolation, the identification of common trends in these metrics and the definition of regression techniques that can be appropriately applied. In addition to the technique, the tool is able to recognize training sets that are problematic for extrapolation and notify the user that manual inspection is required. We show that it is possible to extrapolate well-behaved application address streams with a high-level of accuracy. The extrapolated streams, based on low-level metrics, achieve cache miss rates (a high-level metric) within 1% of the synthetic address streams resulting from PSnAP profiling or are marked as requiring manual inspection.

4.2 Overview of Approach

The goal of this work is to provide a synthetic address stream for application runs that may not be possible to execute on a machine yet. The execution may not be possible because the resource has not been deployed at the core count of interest or profiling the application at that core count is impractical. The synthetic address stream can be used to drive memory simulations or other studies exploring the memory behavior of the application, before it is actually possible to run or profile it at scale.

Table 4.1: The cache hit rates for the most dominant block in FT.

	Cache Hit Rates		
Core Count	**L1**	**L2**	**L3**
4	85.9	8.4	4.5
8	85.9	8.4	4.5
16	85.9	8.4	4.5
32	85.9	11.9	1.7

Address streams capture behavior that is not available in higher level metrics. For instance, a commonly used memory metric is the cache hit rate. Using cache hit rates to predict the memory behavior under strong scaling is problematic. As an illustration, Table 4.1 shows the cache hit rates achieved by the most dominant block of the NPB, FT on a Nehalem cache structure. At a core count of 32 a portion of the working set drops into L2 cache, resulting in a large jump

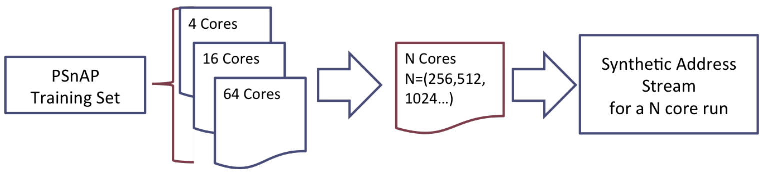


Figure 4.1: Overview of Extrapolation Approach

in cache hit rate (8.4% to 11.9%). Directly extrapolating the predicted cache hit rates at 32 cores from that observed for 4, 8 and 16, by fitting a line or higher-order polynomial will obviously fail. And simply knowing the sizes of the caches will not help either. There is some property of the address stream that changes in a way not captured by the first 3 high order metric measurements (something is changing but you don't see the effect until you hit 32 cores).

We provide the extrapolated synthetic address stream by extrapolating lower-level metrics, specifically the content of PSnAP profiles. PMaC's Synthetic Address Streams from Profiles (PSnAP) are concise representations of address streams resulting from application execution. Each instruction is described as a starting address and a stride pattern. The representation is described in more detail in the next section.

The *extrapolated PSnAP* profile is the result of identifying trends in the recorded behavior of memory address profiles. PSnAP profiles are collected at a series of core counts, the series of PSnAP profiles is referred to as the training set. Through a series of regressions and internal calculations the extrapolated PSnAP profile is generated. That profile can be used to generate a synthetic address stream at the target core count.

The approach as described is meant to aid a user in studying the strong scaling behavior of an application. The approach is described for the cases that it is fully-automated. There are basic blocks in most every benchmark or application where fully-automated extrapolation will fail and manual inspection is required. Fortunately, the tool can identify these cases and greatly reduce the amount of work to be done by the user by automating all of the well structured patterns.

4.3 PSnAP Representation

This section describes the PSnAP address stream representation, referred to as *profiles*. The profiles are human readable. This makes it possible to study the profiles to better understand the access patterns of an application and manipulate the profiles to test the affects of hypothetical compiler optimizations or strong scaling, the subject of this work.

The profiles comprise two sections, first, a representation of instruction-specific patterns. The instruction-specific patterns can be used to replay the address stream resulting from a single instruction. The single instruction address streams are combined in a full-application address stream by following the second section of the profile, the control flow graph. PSnAP stores a compressed form of the control flow graph.

Not all instructions are candidates for PSnAP representation. PSnAP requires that the address streams result from a well-structured address calculation. Such as one that would result from a nested loop structure. Address calculation that are random, or generated by reading data using a gather, are not candidates. Those streams are represented using an alternative representation.

The instruction-specific patterns are expressed with a starting address followed by a stride pattern. This combination allows for the address stream to be replayed. Table 4.2 shows and example instruction-specific pattern. This pattern is taken from the dominant block of the NAS FT benchmark [8]. The *Stride* row lists each of the strides required by the stream. The *Pattern* row indicates how many times the corresponding stride is to be repeated. The *Less Than* row indicates the index of the stride to move to if the number of times we have applied the stride is less than the target (in the pattern row). The *Equal* row indicates the index of the stride to move to once we have applied the stride the target number of times. When this step is taken the state for that index is set to zero.

During replay, the first address is printed. Then the pattern is used to guide the strides which are applied to the previous address printed. Figure 4.2 shows the state of the address stream after 48 steps.

In order to replay the full application address stream the instruction specific

Replay Pattern First Addr: Ox6DC1C0				
Index	**Stride**	**Count**	<	==
0	16	15	0	1
1	48	127	0	2
2	16	15	2	3
3	-36816	16384	0	4
4	16	15	4	5
5	48	127	4	6
6	16	15	6	7
7	-18384	16384	4	0

Table 4.2: The PSnAP profile for the first instruction of the most dominant block in FT.

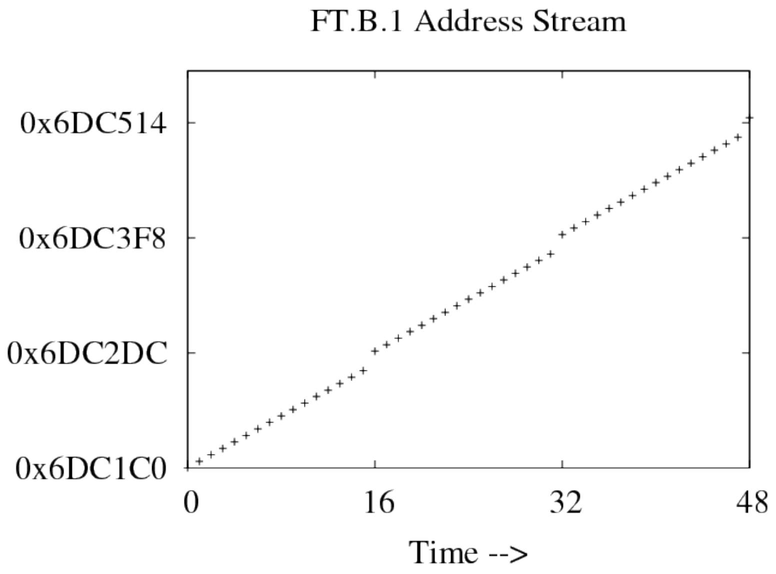


Figure 4.2: The address stream that results from the replay of the FT profile.

address streams must be combined in the same order as during execution. This is achieved by compressing the control flow during recording.

The control flow representation is expressed per loop. Each loop is associated with a set of basic blocks, the top section of Table 4.3 shows that loop 7 has 4 basic blocks. Notice that basic block 565 is special because it is the head of a subloop, loop 8. Loop 8 only has a single basic block. The masks label each block to indicate if they are present in the current execution path. The paths taken through those basic blocks are kept in the path history, in this example only the basic blocks with masks 0 and 2 are executed. The number of times the loop is entered and the number of times it iterates on each entry is kept in the iteration history.

Table 4.3: The control flow representation for the most dominant loop in FT.

	Loop 7				**Loop 8**
ID	563	564	565*	566	565
Mask	0	1	2	3	0

Path History			
Path	**Count**	**Path**	**Count**
0000 0101	4798283776	0000	4798283776

Iteration History			
Iter.	**Entries**	**Iter.**	**Entries**
16	256	1	4798283776
64	64		
256	16		
1024	4		
4096	1		

The path histories indicate that for each of these loops only a single path was taken. The iteration history has been truncated to show one cycle through a repeated pattern. Loop 7 iterates 16 times for 256 entries and then 64 times for 64 entries, this pattern continues until the correct number of visits has been reached. Loop 8 iterates only once for each entry.

The information shown provides the framework for generating a synthetic address stream that displays similar behaviors to that of the observed stream. For extrapolation, the data shown above, the loop-specific stream patterns and the compressed control flow format, are collected at a minimum of three core counts. This data is then used to create a synthetic PSnAP representation at a higher core count.

4.4 Extrapolating PSnAP

The PSnAP profiles are extrapolated to larger core counts using a training set of profiles collected at smaller core counts. The values in the training set profiles are used to drive a least squares regression. Therefore, the training set must contain at least three profiles.

Instruction Specific Patterns

The pattern for each instruction is extrapolated value by value. Table 4.4 shows the training set and extrapolated values for one instruction in the dominant block of LU. The values across all stride lists are extrapolated first. In this example they are constant (eg. 200,200,200 $->$ 200). The same is done for the count list (eg. 49,24,11 $->$ 5). Both of these are done using linear least squares regression(with integer arithmetic). In the case of the count list, least squares regression determines $a = 100$ and $b = -1$ in Eq. 4.2. In this example the core count for the first pair in the training set is X_1 where $2^{1/X_1} = 4$.

$$Y_i = aX_i + b \tag{4.1}$$
$$where : Y_1 = 49, Y_2 = 24, Y_3 = 11, Y_4 = ?$$
$$X_1 = 1/2, X_2 = 1/4, X_3 = 1/6, X_4 = 1/8$$

$$S = \{\text{stride sizes}\}$$
$$C = \{\text{counts}\}$$
$$S_3 = -1 * ((((S_0 * C_0) + S_1) * C_1) + (S_2 * C_2)) \tag{4.2}$$

Table 4.4: The instruction-specific patterns for an instruction in FT at core counts of 4,16,64, and 256.

Replay Patterns I=Index S=Stride C=Count											
4			16			64			256		
Ox6DC1C0			Ox6DC1C0			Ox6DC1C0			Ox6DC1C0		
I	S	C	I	S	C	I	S	C	I	S	C
0	200	49	0	200	24	0	200	11	0	200	5
1	400	49	1	400	24	1	400	11	1	400	5
2	200	49	2	200	24	2	200	11	2	200	5
3	-509600	1	3	-129600	1	3	-30800	1	3	-8000	1

The final value in each stride list is a step back to the beginning of the array. Rather than extrapolating to get it, we do an internal calculation to make sure it is consistent with the previous pattern. Eq. 4.2 shows how to calculate the step back for a pattern of length four. It can be generalized to cover patterns of other sizes as well.

Starting Addresses

Maintaining the relationships between addresses of different instructions is a priority, because of the large impact on performance the relationships may cause. For example, it is possible for two instructions in different basic blocks to operate on the same addresses. The second basic block will likely have low cache miss rates due to this temporal locality, if the relationship is not maintained higher cache miss rates will be observed. Maintaining the relationships requires that the starting address for each instruction be determined using extrapolation.

The extrapolation tool treats each area of virtual memory separately. The addresses in PSnAP profiles are virtual addresses, which are separated into three main areas. The executable sections are located the beginning (or lower addresses) of memory, the heap starts at the conclusion of the executable and grows upward, and the stack starts at the top of virtual memory and grows downward. This arrangement is illustrated in Figure 4.3.

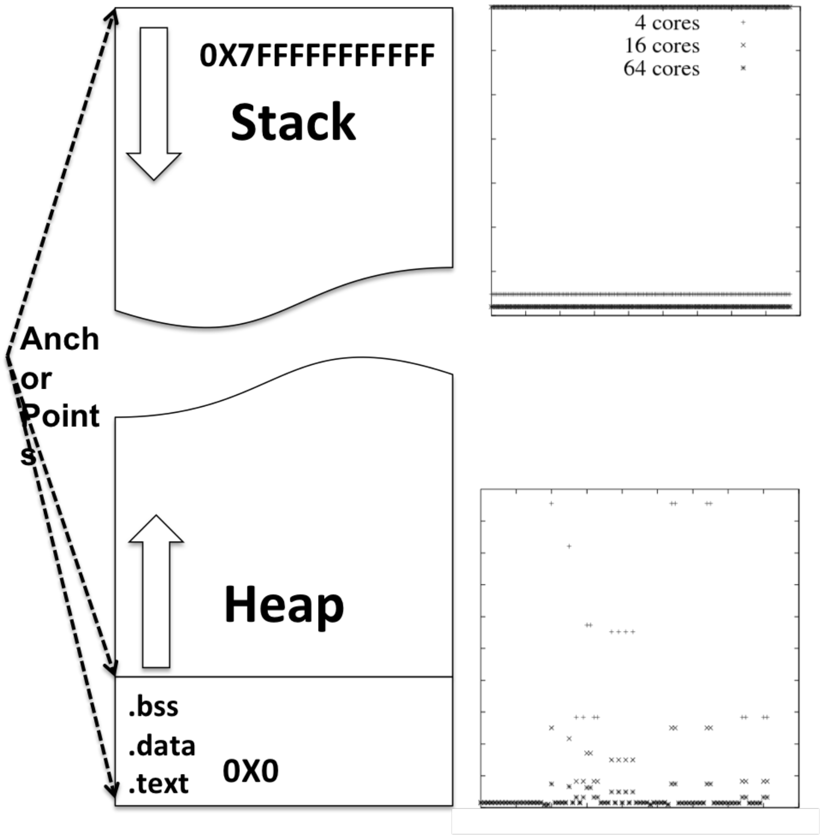


Figure 4.3: Virutal memory and the effect of ASLR

Addresses that fall within the executable are those that are allocated statically. The size of statically allocated data is known at compile-time and it is allocated only one time. These addresses are extrapolated directly.

Addresses that fall on the heap usually result from dynamic allocation due to a call to malloc or related function. Malloc moves through the heap and allocates the memory requested in the first available space of adequate size. As long as the application does not perform a lot of mallocs intermixed with frees the address should be predictable across runs and core counts. These addresses are extrapolated using their distance from the heap pointer, or the beginning of the heap. The heap pointer is estimated by recording the result of an 8 byte malloc in the MPI initialization call.

Addresses that fall on the stack result from data used within functions.

During execution the space used on the stack grows downwards. It is more accurate to extrapolate these addresses with respect to the top of the stack rather than using their actual values. This is because of the magnitude of the values.

However, this is complicated by the use of Address Space Layout Randomization (ASLR). ASLR is a security technique used by Linux that effectively moves the starting address of the stack. The result of this is that across executions and across core counts the starting address of the stack (the stack pointer) changes. The right hand side of Figure 4.3 shows the recorded addresses for three core counts of FT. The effects of ASLR can be seen in the top right hand side. The addresses for each core count are grouped at a different location on the stack, randomly. Therefore, for addresses that fall on the stack, extrapolation is performed using the distance from the maximum address and any properly aligned maximum address can be used as a starting point.

Control Flow Graph

The data in the control flow graph that have the potential to change across core counts are path and iteration histories. These values are extrapolated in two steps. First, extrapolation is attempted on all values using least squares linear regression. Any value that results from an imperfect fit, meaning that the equation determined does not match the training set exactly, is recalculated during the second step. The second step involves performing an internal calculation to determine values that did not extrapolate well.

For example, Table 4.5 shows the control flow scaling for the most dominant loop in LU. The top section is the static loop structure. The middle section is the path history. In this case there is only a single loop, and a corresponding count for each core count in the training set. The lower section is the iteration history. In this example each core count has a pair of iteration configurations. For example, at four cores, one time the loop is entered and iterated 49 times, 25099 times it is entered and iterated 50 times.

The light grey boxes are calculated using linear extrapolation. The dark grey box does not extrapolate exactly and is therefore calculated using an internal

Table 4.5: The extrapolated control flow representation for the most dominant loop in LU.

Loop Structure			
	Loop ID	**Block ID**	**Block ID**
ID	2	1	0
Mask	0	1	2

Path History				
Path	**Count (each core count)**			
	4	16	64	256
0000 0101	1254999	627499	301199	125499

Iteration History							
Iterations (each core count)				**Entries (each core count)**			
4	16	64	256	4	16	64	256
49	24	11	4	1	1	1	1
50	25	12	5	25099	25099	25099	25099

calculation. Let I={iteration counts}, E={entry counts}, and C=path counts.

$$C = \Sigma_{i=1}^{n} I_i * E_i \tag{4.3}$$

The extrapolation of PSnAP profiles is performed in three main sections: instruction specific, starting addresses and control flow. The instruction specific patterns and the control flow data are extrapolated in two steps, direct extrapolation with least squares regression and internal calculations. The starting addresses are determined based on their distance from an anchor, where the anchor points are the beginning of virtual memory, the heap pointer and the stack pointer.

4.5 Experimental Results

Our extrapolation approach was tested using the NAS Parallel Benchmarks. The result of the extrapolation test is that while the automated extrapolation has the potential to be very accurate for well-behaved blocks, it is not a replacement for familiarity with the applications. This is very clearly demonstrated through the extrapolation of FT. FT appears to behave very well, until the core count exceeds the dimensions of the dataset, at that point automated extrapolation breaks down, as does the benchmarks performance.

The accuracy of the extrapolation was tested at a high-level using cache miss rates. The cache miss rates of the extrapolated PSnAP profile were compared against those of a measured profile. The results show a high level of accuracy, the average difference in L1 cache miss rates was low, less than 0.5%.

A manual comparison of the extrapolated and measured profiles was made in order to understand the cache miss rate comparison. This revealed that for a subset of the NPBs the control flow and access patterns were extrapolated with no error (FT, EP); this means that the profiles were examined side-by-side and the patterns were determined to be identical. The cache miss rate for these cases were very accurate, as would be expected. Insight from the manual comparison is used to explain the errors in the cache miss rate comparison, with regard to the control flow and access patterns.

In addition to the cache hit rates an image comparison methodology was employed. The extrapolated and observed addresses for the dominant basic block were used to create figures. The two figures were then compared and the Hausdorff distance between the two computed. This metric gives valuable insight to the fidelity of the extrapolated streams. For instance, it detects, whereas cache hit rates and manual comparison did not, that the starting addresses for the dominant block in FT have some extrapolation error.

The difficulty of extrapolation was explored as well. A subset of the more regular NPBs were extrapolated with no manual effort required, however, a subset of the NPBs required a degree of manual intervention. We measured the level of manual effort required by keeping track of the number of lines edited.

Each of the benchmarks was run on UC San Diego's Triton resource (Nehalem) and profiled using PSnAP at core counts of 4, 16, 64, 256, and 1024. The training sets were each run through the extrapolation software, and the number of questionable patterns was counted. This number is used as a guide to measure whether or not the code has the potential for automated extrapolation. This evaluation led to IS, MG, and SP being disqualified. This does not imply that these codes do not have patterns that can be extrapolated, only that the patterns are quite complex and they require a great deal of manual inspection and direct knowledge of the code base.

As an example, IS was not chosen as a test case for extrapolation. IS is written to work only on a small number of cores and there are different statements executed based on the number of cores. A manual inspection of the code revealed that the code was written to detect the number of cores and change the computation accordingly. This makes IS atypical, and a poor candidate for automated extrapolation.

This section presents the data collected for each of the comparisons performed: cache miss rates, the Hausdorff distance (section 4.5.2), and the level of difficulty in automated extrapolation.

4.5.1 Cache Miss Rates

Table 4.6 shows the results of the cache miss rate comparison. A manual comparison was performed in order to understand the results of the cache miss rate comparison. Essentially, a manual comparison was used to identify patterns with errors in the extrapolation. An actual diff could not be used; a single error could throw off the comparison of the entire file, by shifting the profile down a line.

There is a common trend for the larger levels of cache to have higher error rates. This implies that while the fine-grained patterns are being maintained well through extrapolation, some of the control flow information is not. The control flow, which has a large effect on basic block interactions and temporal locality, is more difficult to extrapolate than the fine-grained patterns. This difficulty comes from the lossiness in the PSnAP profiles. Another possible explanation is that the starting address extrapolation has errors in it that are creating a different set of conflict misses than would otherwise take place. The Hausdorff difference measurements confirm that the starting addresses may be partly to blame for this effect.

Table 4.6: Accuracy of Cache Miss Rates in Extrapolated Stream

Bench Mark Cores Training Set	**% of execution**	**Dominant Basic Blocks PSnAP Original Miss Rate (Absolute Error)**		
		L1	**L2**	**L3**
BT	6.2%	1.6 (0.1)	1.1 (0.9)	1.0 (1.0)
(1024)	6.2%	1.8 (0.1)	1.1 (0.9)	1.1 (1.1)
16,64,256	6.1%	1.8 (0.3)	1.0 (0.6)	1.0 (0.7)
CG	35.8%	7.3 (0.0)	7.3 (0.0)	7.3 (0.0)
(256)	19.8%	5.0 (0.0)	3.9 (0.0)	2.5 (0.1)
4,16,64	7.9%	3.1 (0.0)	3.1 (0.0)	3.1 (0.0)
CG	35.5%	7.3 (0.0)	7.3 (0.0)	7.3 (0.0)
(1024)	19.6%	5.0 (0.0)	3.5 (1.5)	3.0 (2.0)
16,64,256	7.9%	5.0 (0.0)	2.5 (0.6)	2.0 (0.1)
CG	35.5%	7.3 (0.0)	7.3 (0.0)	7.3 (0.0)
(1024)	19.6%	5.0 (0.0)	3.5 (1.5)	3.0 (2.0)
4,16,64	7.9%	5.0 (0.0)	2.5 (0.5)	2.0 (0.3)
EP	55.2%	1.5 (0.0)	1.5 (0.0)	1.5 (0.0)
(1024)	28.1%	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
16,64,256	9.3%	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)
FT	58.9%	13.8 (0.0)	11.3 (0.0)	1.8 (0.0)
(256)	7.2%	14.1 (0.0)	10.6 (0.0)	0.6 (0.0)
4,16,64	5.9%	13.7 (0.0)	12.1 (0.0)	6.4 (0.0)
LU	16.4%	1.7 (-0.8)	0.4 (-0.2)	0.04 (-0.07)
(1024)	16.2%	3.4 (-0.1)	2.0 (-0.1)	0.49 (0.11)
16,64,256	5.7%	1.8 (0.3)	0.3 (-0.2)	0.03 (0.09)

There are three benchmarks with higher than average differences between the miss rates, BT, CG, and LU. BT proved difficult to extrapolate; it required manual effort to figure out patterns. This was due in part to complex and changing control flow patterns as well as complex access patterns.

CG contains random access instructions in its dominant blocks. Each of the instructions eligible for PSnAP representation in the dominant blocks extrapolated with no error, but the random access generator was unable to match the distribution characteristics well enough to get under 1% error.

LU is unique in that there are no stand-out dominant blocks. In fact, LU has to be profiled carefully in order to include the blocks that make up the largest percentage of the execution at 1024 cores in the training set. Two of the blocks that become more dominant in the 1024 case make up less than 1% of the execution in the training set cases. Our extrapolation successfully tracked the growing dominance of these basic blocks as more cores were added.

FT and EP extrapolate with 100% accuracy according to the cache miss rates. Extrapolating FT to 1024 is more problematic, there are changes in the patterns that make automated extrapolation challenging. This change is discussed in section 4.5.3.

Overall the extrapolation proved very accurate. The maximum error observed was 2%, this was in CG within a block that had random accesses. The largest error observed for a non-random block was less than 1%.

4.5.2 Hausdorff Distance

The Hausdorff distance is a metric used to compare images and shapes in graphs. The Hausdorff distance is defined to be maximum distance of a set to the nearest point in the other set [48]. The precise definition is in Eq. 4.4.

$$h(A, B) = max_{b \in B}\{min_{a \in A}\{d(a, b)\}\} \tag{4.4}$$

The goal of using this metric is to quantify the differences between the address streams. Take the address stream for the dominant block of FT as an example. Figure 4.4(a) shows the observed address stream. Figure 4.4(b) shows the predicted address stream for the same portion of the execution. The figures appear to be almost entirely identical. However, when the Hausdorff distance is calculated the result is 16,349 (3.7% of the address space). This difference is larger than one would expect after viewing the figures.

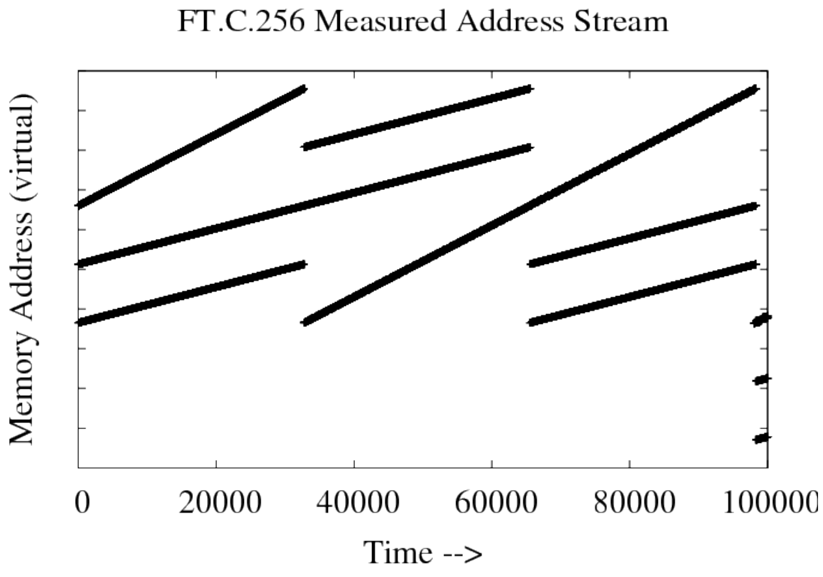
The differences come from the extrapolation of the starting addresses. It is unlikely that this difference would have an affect on the performance of FT, but it is possible that such large errors in the extrapolation of starting addresses would be problematic for applications. Applying this metric, therefore, provides some quantitative feedback on the extrapolation methods that are difficult to ascertain otherwise.

The FT case is the best case in terms of cache hit rate results. The worst case is also FT, but extrapolated to 1024. This case is not presented in Table 4.6 because the extrapolation tool identified that the pattern was breaking down and marked the run as a failure. Both visual inspection of the figures created by running this case and the Hausdorff distance confirm this. The cache hit rates are shown in Table 4.7; they also confirm that the extrapolation failed. The Hausdorff distance for these cases is very high, over 800,000 exceeding 30% of the address space.

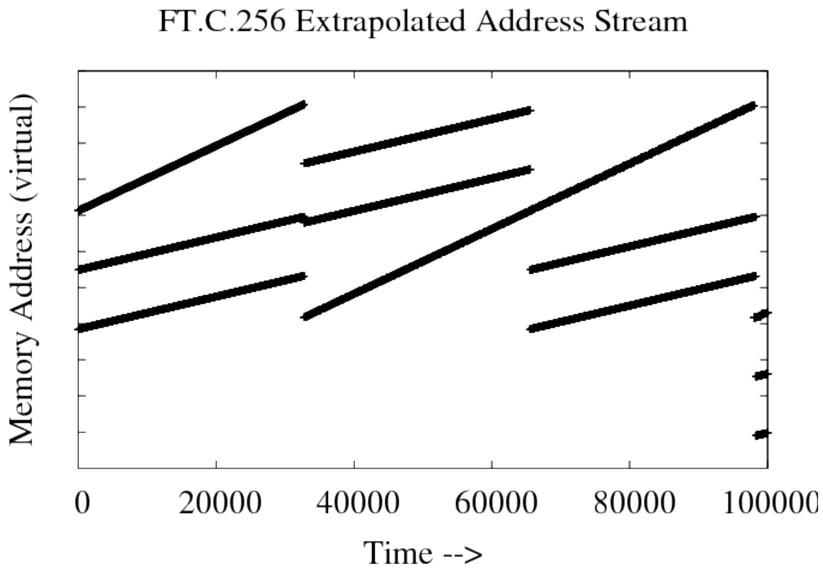
The Hausdorff distance is very expensive to calculate. The time complexity is $O(nm)$ where n is the number of addresses in the extrapolated address stream and m is the number of addresses in the measured. Some pruning is possible to slightly lower the running time, but running Hausdorff for each test case is not reasonable. Fortunately, it is not required.

Any extrapolated stream that can be identified as erroneous through cache hit rates, profile inspection or visual inspection does not require a Hausdorff distance to confirm that it is erroneous. The Hausdorff distance is such a strict measure that it is best applied only to the address streams that appear to be the most accurate when evaluated through other terms.

The Hausdorff distances were calculated for EP, FT and CG. FT showed the highest error at 3.7% of the address space; both FT and CG had Hausdorff distances that made up less than 0.1% of the used address space.

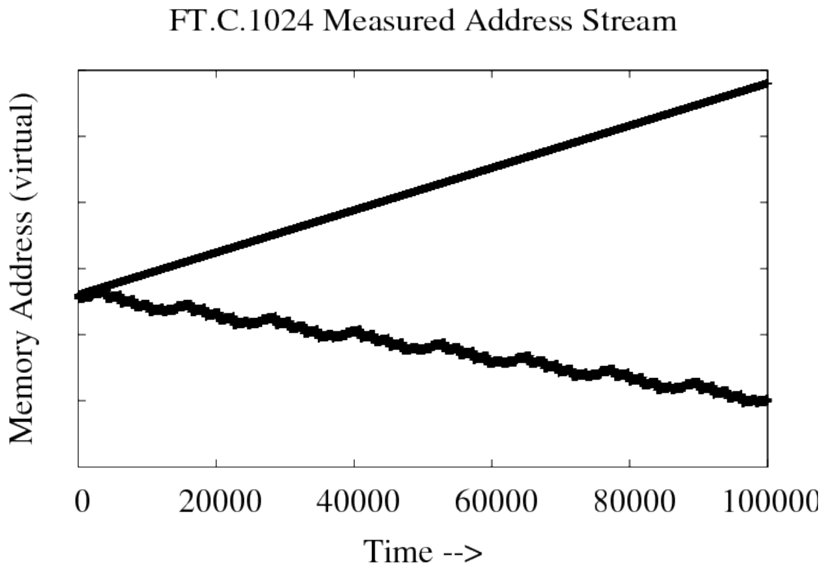


(a) Address Stream

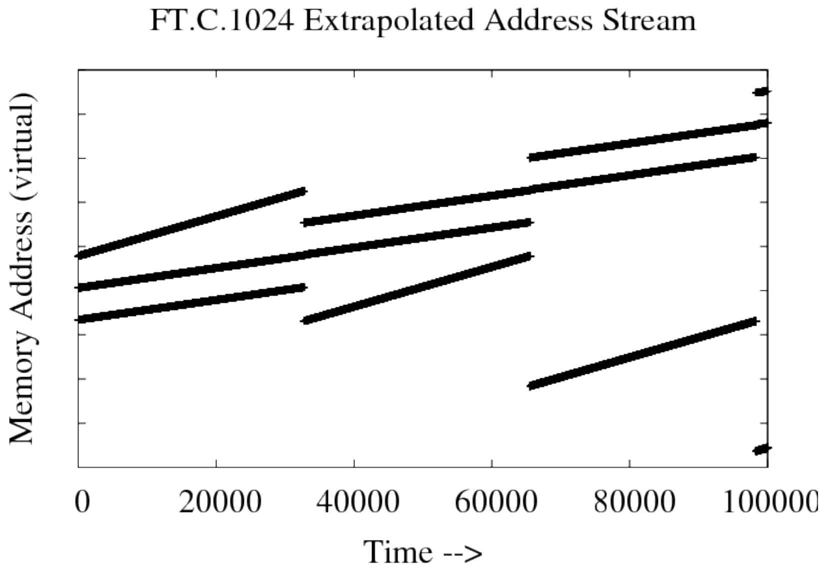


(b) Extrapolated Address Stream

Figure 4.4: A comparison of the observed and extrapolated address streams for the dominant block of FT at 256 cores.



(a) Address Stream



(b) Extrapolated Address Stream

Figure 4.5: A comparison of the observed and (failed) extrapolated address streams for the dominant block of FT at 1024 cores.

4.5.3 Manual Effort Required

Table 4.8 shows the number of lines for each extrapolation that required manual intervention. The manual inspection is required because of a major change in the access pattern detected within the training set. As an example, when extrapolating FT using a training set of 4, 16, 64, and 256 there is a shift in the access patterns between 16 and 64. The extrapolation software cannot handle this shift, however, it is possible for a user to make a reasonable guess. Table 4.9 displays the data containing this shift. Essentially, the core counts of 4 and 16 are not good choices for the training set for this instruction.

BT, and SP required the most manual work. Each of these perform a great deal of communication and, therefore, coordination with other cores. This appears to complicate the control flow, making the extrapolation process more complex. Codes such as FT and LU are more regular and, therefore, can be extrapolated with very little manual work. CG was quite easy to extrapolate as well, though, the large “random” access pattern in the dominant basic block made full extrapolation accuracy difficult.

The benchmarks that required manual effort to extrapolate, did not require as much as Table 4.8 implies. Most of the patterns that resulted in errors were repeated, for instance in BT only 31 unique patterns were manually extrapolated.

4.6 Conclusion

We presented a method of extrapolating PSnAP profiles in order to study the memory performance of an application under strong scaling. A series of PSnAP profiles, referred to as the training set, is used to identify trends in the instruction-level access patterns and control flow of an application address stream. The trends are identified and used to create an extrapolated PSnAP profile, which can then be used to create a synthetic address stream at the new core count.

This form of automated extrapolation for strong scaling is not a substitute for in-depth application knowledge. It is a tool that can greatly reduce the effort required by a user to perform a strong scaling study. The tool extrapolates

Table 4.7: Accuracy of Cache Miss Rates in Extrapolated Stream

Bench Mark Cores Training Set	**% of execution**	**Dominant Basic Blocks PSnAP Original Miss Rate (Absolute Error) L1**	**L2**	**L3**
FT	39.0%	50.0(35.5)	19.3(5.5)	8.5(-4.2)
(1024)	7.6%	3.4(1.3)	0.8(-1.3)	0.1(-1.8)
16,64,256	6.5%	0.0(0.0)	0.0(0.0)	0.0(0.0)

Table 4.8: A summary of the extrapolation process.

Bench Mark	**Training Set**	**Core Count**	**Direct Extrapolation**	**Internal Calculation**	**User Decision**
BT	16,64,256	1024	14,568	406	305
CG	4,16,64	256/1024	1796	0	8
	16,64,256	1024	1803	0	10
EP	4,16,64	256/1024	226	0	7
	16,64,256	1024	219	0	0
FT	4,16,64	256/1024	1839	8	0
	16,64,256	1024	1839	8	0
LU	4,16,64	256/1024	8755	730	1
	16,64,256	1024	8745	740	1
SP	4,16,64	256/1024	29343	957	650
	16,64,256	1024	29655	1009	232

well-behaved patterns accurately and identifies those that require further manual inspection.

The accuracy of the extrapolation was evaluated in three ways. First the achieved cache miss rates of the measure and extrapolated address streams were compared. The extrapolated streams display high fidelity, achieving accuracy above 99% for the instructions that do not contain random sections. Next the streams were used to generate figures that were compared by calculating the Hausdorff distance between them. This metric revealed differences between the starting addresses of patterns that were not apparent in the cache hit rate evaluation. Lastly, the difficulty of extrapolation for each benchmark was explored. Even for the most difficult benchmarks, the tool was able to greatly reduce the number of patterns that needed manual inspection.

Table 4.9: An example problematic instruction from FT.

Replay Patterns I=Index S=Stride C=Count								
	4		16		64		256	
	Ox34103020		Ox30F75410		Ox151EB000		Ox15ED1050	
I	S	C	S	C	S	C	S	C
0	288	511	288	511	288	511	288	511
1	-147152	15	-147152	15	-147152	15	-147152	15
2	288	511	288	511	288	511	288	511
3	-147408	1	-147408	1	-147408	11262	-147408	2814
4					288	511	288	511
5					-147152	15	-147152	15
6					288	511	288	511
7					492976	1	377568	1

Chapter 5

Common Work Group Discovery

5.1 Introduction

Memory behavior extrapolation using PSnAP profiles as described in the previous chapter requires a mapping step between core counts. The PSnAP profiles encapsulate the memory behavior as observed on each core. In the case of the NPBs the profiles have only very minor differences for each core. However, the majority of large-scale HPC application have a much more complex data decomposition problem than represented in the NPBs. In the case of real applications the work is spread across all of the available processors in varying patterns depending on that decomposition. In these cases extrapolation requires not only extrapolating fine-grained patterns, but organizing the profile from each rank of the execution into groups.

The goal of this work is to identify the profiles from a full application trace that can be used as the training set for PSnAP extrapolation. Only a single profile is required from each common work group. An overview of our approach is outlined in Figure 5.1; the steps described here are coloring the profiles and tracking the groups across core counts. Coloring the profiles refers to grouping the profiles, within each set of profiles. The profiles are colored based on their control flow behavior. Tracking the groups refers to associating groups across core counts based on similar behavior. Once the coloring and tracking is complete the training sets can be given to the extrapolation process.

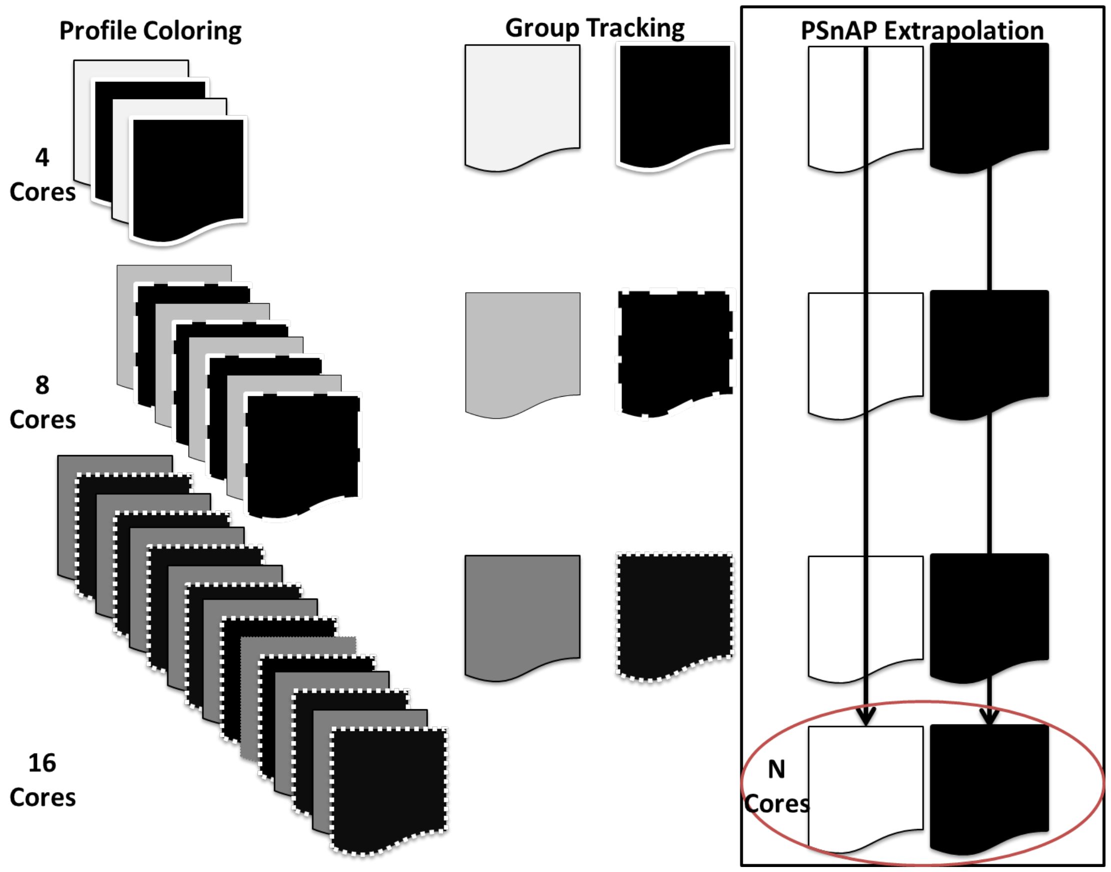


Figure 5.1: An overview of statistical clustering's role in performance extrapolation.

Coloring the profiles or grouping ranks that perform similar types and amount of work into working groups is done using statistical clustering. Statistical clustering has been used by several projects in the past. We use a slightly modified version of statistical clustering based on basic block visit counts to identify groups and map the groups between core counts.

There are limitations to this approach. Applications, such as some weather simulations, break up work in such a way that tracing work groups through processor count changes is not straight forward. Extrapolation of such application requires a less direct approach and is the subject of future work. However, applications with well-structured code decompose in a predictable manner that allows for automated mapping.

This section explains the process of coloring and tracking working groups.

Three full-scale applications where chosen to demonstrate the effectiveness of our grouping method, Avus, Lammps, and PFlotran. Each of these applications are large-scale memory-intensive scientific applications that are commonly run on government supercomputing resources and whose time to-solution for fixed problem sizes is important.

5.2 Approach

Parallel applications assign tasks to each of the processors available to them. Often there are patterns to these assignments and groups of processors that perform similar work[42, 49]. Identifying these *working groups* enables a trace to be identified that represents the behavior of the entire group. The number of working groups varies depending on the application and implementation. This work aims to identify the smallest number of working groups that can accurately represent and application.

A basic block vector(BBV) is a representation of the number of times each basic block within an application execution was visited[51]. By comparing the BBV associated with each processor during an execution it is possible identify the working groups, a process referred to as clustering. The following sections discuss the process by which the BBVs are collected, pruned and clustered, and a representative trace is created.

5.2.1 Collecting and Pruning the BBV

The BBV is collected using PEBIL [35] and consists of a list of basic blocks along with the number of times that basic block was visited during an execution of the application. A large scientific code can consist of thousands of basic blocks. Avus, as run during tracing for this work, produced 23,247 basic blocks and LAMMPS produced 34,982. Using this number of basic blocks to determine working groups would be very difficult. First the process would be extremely slow due to the size amount of data and second the process is complicated by noise in the data associated with basic blocks that are not contributing significantly to the

Table 5.1: The Reduction in Basic Block Count Due to Pruning

Application	**Basic Block Count**	**Reduced Block Count**
Avus	34,982	298
LAMMPS	23,247	43
PFLOTRAN	29,576	133

performance behavior.

The PMaC Prediction Framework [55] has long been pruning the list of basic blocks to be instrumented by selecting only those blocks in the top 95% by contribution to dynamic operation count. Essentially the basic blocks from each processor are sorted by number of instructions executed. Those blocks in the top 95% for each processor are selected to be instrumented. This method of pruning is used to reduce the BBVs to a manageable number, without losing pivotal data.

5.2.2 Identify Working Groups

Once the BBVs have been effectively pruned they can be used to identify behavior groups using the k-means method of clustering algorithms. A large body of work has been done in clustering algorithms. A discussion of alternative algorithms can be found in Jain and Dubes [28]. K-means attempts to identify center of natural clusters in the BBVs by minimizing the total intra-cluster variance. Harigan and Wang [26] provide a more detailed description of the method.

This method of clustering was chosen because the algorithm is very fast and while the accuracy of the solution is very dependent on the input, the speed allows for multiple runs to determine the best clustering. The statistical clustering performed based on the BBVs used clustering software package, CLUTO [30]. CLUTO provides a variety of clustering algorithms as well as as providing variance statistics in order to aid in the best cluster choice decision.

An example of using statistical clustering on the BBVs for a 64 processor run of the application LAMMPS [47] (a molecular dynamics application) can be seen in Figure 5.2. In this figure the computation time (e.g. time on processor between communication events) is plotted for each processor of the run. The statistical clustering identified two processor groups from this group of BBVs. It illustrates that first, the behavior groups cannot be identified by computation time alone, and second, that there is a clear pattern to the processor organization in LAMMPS which statistical clustering is able to identify in an automated fashion.

K-means was run ten times for both Avus and LAMMPS. The sum of squared errors was used within each cluster to determine the validity of the clustering. For LAMMPS that error dropped dramatically when moving from a single cluster to two clusters and dropped only slightly as additional clusters where added. The sum of squared errors within clusters for Avus did not demonstrate the same initial drop and was therefore determined to have only a single behavior cluster.

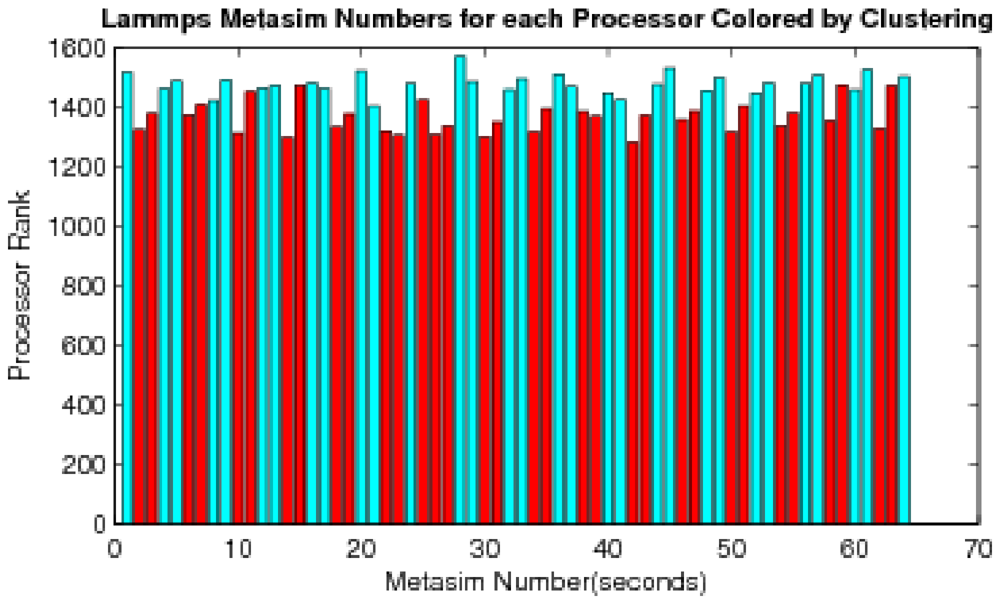


Figure 5.2: A clear pattern of similar behavior groups emerges when the statistical clustering algorithm is run on a set of BBVs created for the LAMMPS application (64 processors)

5.2.3 Creating a representative trace

After processor groups are determined using statistical clusters a single trace file is created to represent the behavior of all the traces in the group. This trace file is augmented with the numerical mean, standard deviation, maximum and minimum across the group for each value recorded in the trace file. This allows for further verification that there were no obvious issues with the canonical representative trace chosen by k means. Representing each behaviorally similar group by a single trace file in this way significantly reduces the number of traces to deal with and therefore the complexity of the extrapolation.

5.3 Mapping Groups Between CPU Counts

The end goal of this process is a series of N numbers, where N is the size of the training set, for each metric within each basic block. At this point all of the values have been determined, but it is still necessary to group them into a series. Refer back to Figure 5.1, profile coloring is complete, but how is it determined that the trace represented by the dark files with dashed outlines are representing similar working groups at different processor counts?

Identifying which groups are performing similar work can be done using the BBVs of their traces. There are several ways to approach this. The most simple is to replace the visit counts for each basic block with a one and then apply the k-means algorithm. The assumption is that the traces that represent the same work will have visited the same basic blocks, just a different number of times. This may be accurate for some applications, but a more robust method would be to normalize the data to fit in a specific range and apply k-means.

Once the mapping for behavior groups at each processor count has been made it is trivial to extract a series of data, composing the training set for each metric within each basic block.

5.4 Conclusion

Common work group discovery is an important component of PSnAP extrapolation. Fix structured applications can be handled using simple automated approaches depending on k-means clustering. More dynamic applications require direct user knowledge.

This chapter presented an automated approach to common work group discover and tracking. A software infrastructure to perform such actions has been developed as well.

Chapter 6

Related Work

Several categories of work are pertinent to the PSnAP compression and application scaling techniques presented here. Memory subsystem simulation and simulation-aided performance modeling are the primary consumers of application address streams and the driving force behind the improvements in their collection and handling. These improvements include alternatives to storing the streams, synthetic stream generation, stream characterization and stream compression. The strong scaling behavior of HPC applications has also been examined. This section outlines some of the most pertinent research in these areas.

6.1 Memory Simulation

The primary consumers of address streams are memory subsystem simulation and simulation-driven memory performance studies. For this reason many of the advances in stream collection and characterization have come out of these projects. The following is a brief overview of projects using memory simulation.

Memory performance has been the focus of rigorous research in HPC for several decades. Even before the term "Von Neumann bottleneck" was coined by John Backus in his 1977 ACM Turing award lecture [6], a great deal of work was being done to evaluate and predict the performance of computers with a strong focus on memory subsystem performance [4, 3, 12]. As early at 1967 Chu Ping Wang et. al at IBM Watson began working with address streams and cache simulators as a method for performance evaluation[3]. Another example of simulation based memory hierarchy evaluation, also from IBM is Mattson et. al in 1970 [37]. They described using cache simulation to evaluate the effectiveness of different cache structures and policies.

More recently a great deal of work has been done to create cross-architectural performance predictions. This type of prediction involves measuring specific traits of an application on one machine and using that information to predict the application's performance on a different resource, referred to as the target resource. This is an important line of research because the target resource may not be deployed.

The PMaC Framework [55] is currently being used by the department of defense's high performance modernization office (HPCMO) to perform cross- architectural predictions. The predictions inform acquisition decision during each acquisition cycle; aiding the HPCMO in selecting resources that are well-suited for their specific HPC workload.

The PMaC Framework is designed to observe an application address stream and record metrics about the execution in an *application signature*. The target machine is measured using benchmarks and that data is recorded in a *machine profile*. The application signature and machine profile are combined during a process called convolution. Convolution involves both simulation and analytical modeling.

During the observation of the memory address stream two techniques are used to avoid saving the address stream. First, PMaC employs on-the-fly processing; this means that the address stream is observed and processed as it is generated, the stream is never saved. Second, PMaC uses aggressive sampling; sampling involves turning off the instrumentation code that observes and processing the stream for periods of time in order to avoid high overheads.

This on-the-fly processing is a very effective means of handling large address streams. The drawback is that any time the simulation parameters change the execution must be repeated on an HPC resource. PSnAP would enable the PMaC framework to save and re-process address streams after the fact.

Another prediction framework is Prophesy developed at Texas A&M by Valerie Taylor [69]. Prophesy involves automated static and dynamic instrumentation as well as user level static analysis in the preparation steps before automated modeling begins. The data collected by the instrumentation steps is collected into a database and through querying those results performance predictions are created. Prophesy also employs on-the-fly processing to avoid the collection of full address streams.

6.2 Address Stream Processing

Processing the address stream involves observing the address stream and then recording some aspects of the stream. As has been discussed recording each of the addresses is not a reasonable approach. Several other approaches have been taken in order to collect an useful amount of information without the process becoming intractable. These approaches include avoiding saving the address stream entirely by collecting high-level metrics on-the-fly, characterizing the address stream in order to generate synthetic streams later, and compressing the address stream into a useable representation.

Any effort to process an application address stream starts with accessing and observing the stream. This is typically done by instrumenting the binary. The instrumentation code must not perturb the address stream as it records it, must add as little overhead to the execution as possible, and must use a minimal amount of memory or execution will fail. PSnAP uses a binary rewriting tool called PEBIL [35]. PEBIL works on the Linux/X86 instruction set and maintains a competitive overhead when compared with other binary rewriting tools. PEBIL is based on a previously developed tool called pmacinst [62]. This tool worked exclusively on the Power series instruction set.

Another widely used instrumentation tool is valgrind [41]. Valgrind takes a unique approach to binary instrumentation that allows for low overheads. It is possible to implement PSnAP using any of these tools as PSnAP is a library that plugs into the binary instrumentation tool.

6.2.1 On-the-fly Processing

On-the-fly processing is commonly used in performance modeling to run a cache simulator. The addresses generated by an application are collected into a buffer and the buffer is sent through the cache simulator when its capacity is exceeded. PMaC uses this approach to populate its application signatures. The cache simulation data is independently valuable, another, widely available tool that can perform this task is cachegrind.

Cachegrind [41] is a tool built within valgrind that uses binary instrumentation to perform cache simulations. This tool is used by software developers to evaluate their implementations for specific cache structures. It also supports memory performance research. Cachegrind simulates one cache structure on-the-fly during execution and incurs a slowdown of approximately 40X for the NPBs. This is a very small slowdown for a cache simulation tool with no sampling. It may be possible to improve the overhead of PSnAP by implementing it within the valgrind framework.

On-the-fly processing incurs a high slowdown overhead. A slowdown of 1000X is not uncommon. Even with a slowdown of 40X, which cachegrind achieved, or 90X, which PSnAP achieved, running HPC applications is challenging and sometimes impossible. For this reason, on-the-fly processing is typically coupled with aggressive sampling. Coupled with aggressive sampling the slowdown can be reduced to under 10X.

6.2.2 Characterization

Also directly related to our work are other efforts to summarize application behavior in order to create synthetic address streams. Mattson [37] makes clear that at that time (1970) it was already common practice to use synthetic address streams which were assumed to represent current computational workloads in order to drive cache simulators as a method of memory subsystem evaluation. Some of the streams used for performance evaluation were generated by simple methods including random number generation and it was shown by Smith et. al that this method is not adequate and the choice in workload greatly affected the outcome of the evaluation [54, 53].

Very early attempts at creating synthetic address streams depended on random number generators and other mathematical means, but most attempts depend on metrics taken from application or benchmark executions. This work draws on the experiences of other groups who have attempted to classify program behavior based on the locality characteristics of their address streams [11, 16, 2, 68, 60, 24, 9].

Sorenson et. al [57] expanded on work done by Grimsrud et al [24] that demonstrated that none of the five well-known approaches to this area achieved a high level of accuracy. The general categories examined are: the Independent Reference Mode (IRM), the Stack Model, the Partial Markov Reference Model, the Distance Model, and the Distance-Strings Model. Each of these was shown to not preserve the access patterns and locality demonstrated in the original trace. IRM came the closest to achieving this, but missed important features of the trace.

Even though other groups have attempted to classify program behavior based on the locality characteristics of their address streams [11, 60, 16, 9, 68, 2, 24], achieving within 90% verisimilitude when using these streams as representatives of the full application to predict cache hit rates has not been possible until a project called Chameleon [66]. In Chameleon, the granularity used during analysis for the profiles is quite different than that used here. Chameleon focuses on stream and application as a whole, PSnAP breaks them down into their constituent pieces. This change in granularity allows for the extrapolation of address profiles that was not previously possible.

6.2.3 Compression

PSnAP is a lossy compression technique specifically tailored to work with HPC address streams. It is not the first attempt at taking advantage of regularities in address streams for compression. The following discussion addresses previous work done in both the areas of address stream compression and synthetic address stream generation. PSnAP differentiates itself from all of the work described below because of the granularity of analysis, the compression ratios achieved, the overhead of collection and replay, and the fact that the profiles are human readable and lend themselves to manipulation.

Address streams prove to be extremely well-suited for compression. They contain patterns that can be recognized by most modern compression schemas due to regularity found in most applications. Gzip has proven to achieve about two order-of-magnitudes reduction in size[22], which when considering the size of the traces that result from HPC applications is not enough to create manageable file sizes. Gzip is a generic compression schema, not developed specifically for use with address streams.

Several schemes have been developed specifically for address streams and they are able to achieve much higher level of compression, up to six orders of magnitude (Sequitur [39], VPC [15], Mache [50], and SIGMA [17]). These schemas have two main disadvantages. First, the time required to perform compression is long, and second the compression ratio is unpredictable because it depends on finding regular patterns in the address stream and treats all streams the same, even when the desired patterns do not exist. Each of these methods is lossless, meaning that even areas of random accesses are saved. Conversely, PSnAP quickly identifies instruction streams that do not contain repeated patterns and employs a different characterization scheme, therefore, not saving random data. Attempting to compress random accesses takes a large amount of time, and in most cases the compression ratio is insignificant.

Sequitur is the standout performer of this group. It works by creating a context free grammar based on the patterns repeated in the address stream. The grammar is created dynamically during compression. However, Sequitur fails to work for some address streams, for example compression of the EP address stream failed with Sequitur [21].

A lossy compression technique was presented by Gao et al [21] referred to as Path Grammar Guided Trace Compression (PGGTC). PGGTC works in a very similar fashion to Sequitur with the exception that rather than creating a context free grammar on-the-fly it uses static analysis to build a control flow graph, which can be used to create a context free grammar. Gao also realized that some portions of an address stream are truly random and therefore do not lend well to compression. Rather than attempting to compress them, they are detected, summarized and regenerated. This summarization is what makes this compression technique lossy.

PGGTC and PSnAP are similar in that they both employ a statically generated control flow graph to aid in the control flow compression. The differences are found in the handling of the addresses. PGGTC depends on algorithm techniques such as Lempel-Ziv and Sequitur to compress the address streams in each basic block. The PSnAP technique separates the address streams by one more level and uses a compression technique developed specifically for address streams.

ScalaMemTrace [10] is a recently developed tool with the same high-level goals as PSnAP. It uses a per-instruction representation to save the patterns. Their representation is a hierarchical structure of Regular Pattern Descriptors (RSDs). The descriptors can be combine to save patterns in a manner similar to the stride histories used by PSnAP. The online collection algorithm used by PSnAP has a lower time complexity than ScalaMemTrace. ScalaMemTrace is O(MS) where M is the number of unique addresses in the stream and S is the number of addresses in a processing window. PSnAP does not require the use of a window for processing and the time complexity is dependant on the number of strides rather than the number of addresses (O(NS) where N << M). As an illustration, the number of addresses in a stream will be approximately the size of memory used divided by the size of the data type, typically in the hundreds of thousands. In the NPBs the maximum number of unique strides for a pattern was 15.

6.3 Application Scaling

Gaining insight to the performance and scaling behavior of applications has long been understood to be an important issue. Several projects have focused primarily on a single application or class of applications [63, 31, 58]. This work is particularly worthwhile when focusing on applications that are run repeatedly on high-end resources, however not all projects have the time or funding to commit to this type of modeling evaluation. For that reason several automated frameworks have been developed.

Several systems have been developed that focus primarily on scaling predictions. BigSim [71] simulates applications running on thousands of processors. It was developed by researchers at the University of Illinois at Urbana-Champaign. The idea is to emulate an application running on some number of processors on a machine that has many fewer processors available. Using that trace, BigSim then simulates the event trace and is able to predict performance on the larger, perhaps unavailable machine. The traces collected by BigSim are event traces which can be quite large and require replay on simulators that include processor simulations and network simulations. Due to the full-scale replay, the slowdown is very large and for many applications the memory behavior dominates the performance. Focusing primarily on memory behavior, avoids some of the slowdown while still providing accurate performance predictions.

A related approach to strong scaling analysis was attempted by Mendes et. al [38]. Their approach depended on the assumption that as an application scaled the behavior of each code section depended on a defined variable, N. By analytically expressing the performance of each section using a function of N, the performance could be predicted as N changed.

Closely related to this work is a project being worked on at the University of Rochester by Chen Ding et. al [72]. Their method attempts to predict the miss rates for an address streams across all program inputs for applications based on extrapolation of the reuse distances found in address streams. This is not a cross-architectural prediction. The cache structures remain unchanged and the address stream is adjusted to emulate the changes that would occur given input changes to the application.

The use of statistical models to represent sequential execution blocks with BigSim was proposed in [70]. This is similar to our work in that the models are directed at small units of code rather than the application execution time as a whole. Pablo and several related projects [42],[49] have utilized statistical clustering as well. Pablo used statistical clustering on the fly during trace collection to reduce the size of the traces.

Acknowledgements

Chapter 6 contains material reprinted from the dissertation authors research exam for the Department of Computer Science and Engineering, University of California, San Diego. *Simulation guided performance modeling.* 2008. [45] The dissertation author was the sole author of this paper.

Chapter 7

Conclusion

7.1 Summary

Due to the widening performance gap between processors and memory, memory performance is the dominant factor in computational performance in high performance computing. Memory address stream driven simulation studies are an important tool for exploring memory performance, both from the perspective of the hardware designer and application developer. This work presents a compression scheme that enables the storage, sharing and manipulation of previously unmanageable address streams.

PMaC's Synthetic Address Streams through Profiles (PSnAP) is presented in chapter 3. PSnAP is a compression technique specifically designed for HPC application address streams. PSnAP breaks the address stream down into constituent components based on program structure. Address stream patterns are identified on a per-instruction basis and the instruction streams are organized into basic blocks and loops. A separate control flow compression technique is used to replay the instruction streams in the correct order.

The synthetic streams generated from the PSnAP profiles are evaluated for accuracy using a cache simulator. The cache miss rates of the observed address stream are compared to those of the synthetic address streams across a variety of cache structures. The synthetic streams match the observed streams to within 1% absolute error for all of the benchmarks that are identified as candidates for

PSnAP compression.

Chapter 4 presents a use case of the PSnAP profiles. Strong scaling performance is explored by using a series of profiles collected for the same benchmark at varying core counts. Patterns and trends are identified within the profiles and are extrapolated to create a profile at a higher core count which is subsequently used to generate a synthetic stream.

The NAS Parallel Benchmarks are used to demonstrate the effectiveness of the compression technique and the extrapolation of PSnAP profiles. These benchmarks represent computations that are common components of HPC applications and encountered in practice. The program structure of the benchmarks are simpler than most HPC applications in one specific aspect. Each of the cores assigned work is doing the same thing on different (or identical) data. Real applications are broken in to working groups.

In order for the extrapolation to be effective on HPC applications working groups must be identifiable in an automated fashion. Chapter 5 presents a method for discovering working groups in real world applications. The method is performed on three applications.

7.2 Future Work

The work presented here has valuable applications in the field of HPC performance modeling. For this reason, a great deal of future work is planned for the further development of PSnAP. In addition to improvements on PSnAP, the format, and the fact that is can be directly manipulated has opened up the possibility of new research projects related to auto-tuning and power consumption modeling. The following is a list of planned software-related improvement and research projects stemming from the foundational work on PSnAP.

7.2.1 Disqualifying Instructions

There are instructions that are not good candidates for the PSnAP compression technique. Instructions that generate a random or semi-random address stream with no recognizable pattern are not candidates. Recognizing these instructions can be done one of several ways.

The current implementation of PSnAP uses a hard limit on the number of strides to be recorded per instruction. For the NPBs this limit was set to 20, and was never reached for an instruction that was a good candidate. There are patterns in real applications that will contain more strides than that and choosing a hard limit may prove to be challenging and clumsy.

A more efficient way to detect a random stream is to disqualify instructions that do not generate any repeated strides early in the pattern. All of the patterns observed thus far have immediately begun repeating strides, within the first 3 addresses. A very simple test is to check the first three accesses for a repeat and disqualify any without a repeat. It is possible to create an address stream that breaks this test, but such a stream has not been observed in benchmarks or real applications.

7.2.2 Sampling Potential

PSnAP does not currently use any sampling, but sampling is planned as a future addition. As it is the only way to circumvent the theoretical minimum overhead. The theoretical minimum is the slowdown incurred by collecting the addresses into a buffer and then looping through the buffer.

One of the fundamental approaches to speeding up any kind of analysis on address streams is to implement sampling. Most tools implement what is called random interval sampling. In this scheme some number of addresses are observed and then instrumentation is turned of for another interval of addresses. Another scheme is to turn off profiling after a maximum number of samples is observed. The PMaC framework combines approaches, sampling only approximately 7% of the overall address stream, while maintaining accurate results.

The problem with these types of sampling is that PSnAP is recording temporal properties within the stream and may miss whole phases of execution that introduce new strides. PSnAP requires a more tailored form of sampling.

The currently proposed solution is to turn off profiling for all instructions in a basic block with the exception of one. The profiling is to be turned off after no new strides have been encountered for a set period of time. One instruction remains in profiling mode in order to detect when a new stride is encountered. At that point profiling for the entire block would be turned back on.

7.2.3 Auto-tuning

Auto-tuning refers to optimizations performed on an application in order to maximize performance by another program, such as a compiler. Auto-tuning is difficult in part because there are many optimizations available and each of those optimizations has parameters. For instance, loop unrolling may be a good choice for some loops, but how many times should the loop be unrolled? In addition, the order that optimizations are applied has a large impact on performance. The combination of these factors leads to a very large space to search when trying to find the best solution for a given application.

It is possible to manipulate the PSnAP profiles generated for a loop in order to explore the search space without actually re-compiling and executing the code. The PSnAP profiles also reveal which compiler optimizations have already been applied. This is at times a difficult task to perform, when the given compiler does not provide feedback.

Bibliography

[1] AFRL. http://www.erdc.hpc.mil/hardsoft/software/avus. 2008.

[2] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, 1989.

[3] W. Anacker and C.P. Wang. Performance evaluation of computing systems with memory hierarchies. *Electronic Computers, IEEE Transactions on*, EC-16(6):764–773, Dec. 1967.

[4] William Anacker and Chu Ping Wang. Evaluation of computing systems with memory hierarchies. *IEEE Transactions on Electronic Computers*, EC-16(6):670–679, December 1967.

[5] K. Auyeung and P. W. Dowd. Synthetic address trace generation for multiprocessor systems. In *in Proc. Western Simulation Multiconference*, 1993.

[6] John W. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[7] D. H. Bailey and A. Snavely. Performance modeling: Understanding the present and predicting the future. In *EuroPar*, 2005.

[8] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[9] Kristof Beyls and Erik H. D'holl. Reuse distance as a metric for cache behavior. In *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, pages 617–662, 2001.

[10] Sandeep Budanur, Frank Mueller, and Todd Gamblin. Memory trace compression and replay for spmd systems using extended prsds? *SIGMETRICS Perform. Eval. Rev.*, 38:30–36, March 2011.

[11] Richard B. Bunt and Jennifer M. Murphy. The measurement of locality and the behaviour of programs. *Comput. J.*, 27(3):238–253, 1984.

[12] Peter Calingaert. System performance evaluation: survey and appraisal. *Commun. ACM*, 10(1):12–18, 1967.

[13] L. Carrington, D. Komatitsch, M. Laurenzano, M. Tikir, D. Michea, N. Goff, A. Snavely, and J. Tromp. High-frequency simulation of global seismic wave propagation using spcfem3d_globe on 62k processors. In *Supercomputing*, 2008.

[14] P. Colella. Defining software requirements for scientific computing. In *DARPA HPCS presentation*, 2004.

[15] Martin Burtscher Computer. Vpc3: A fast and effective trace-compression algorithm. In *IEEE/USP International Workshop on High Performance Computing*, 1994.

[16] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, 1972.

[17] L. DeRose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia. Sigma: A simulator infrastructure to guide memory analysis, 2002.

[18] Peter Kogge et al. Exascale computing study: Technology challenges in exascale computing study: Technology challenges inachieving exascale systems. Technical report, DARPA, September 2008.

[19] J. Flanagan, B. Nelson, and G. Thompson. The inaccuracy of trace-driven simulation using incomplete multiprogramming trace data. In *MASCOTS*, 1996.

[20] X. Gao, M. Laurenzano, B. Simon, and A. Snavely. Reducing overheads for acquiring dynamic traces. In *International Symposium on Workload Characterization*, 2005.

[21] X. Gao, A. Snavely, and L. Carter. Path grammar guided trace compression and trace approximation. *International Symposium on High-Performance Distributed Computing*, 0:57–68, 2006.

[22] X. Gao, A. Snavely, and L. Carter. Path grammar guided trace compression and trace approximation. In *International Symposium on High Performance Distributed Computing*, 2006.

[23] Xiaofeng Gao. *Reducing time and space costs of memory tracing*. PhD thesis, University of California at San Diego, La Jolla, CA, USA, 2006.

[24] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. On the accuracy of memory reference models. In *the international conference on Computer performance evaluation : modelling techniques and tools*, pages 369–388, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[25] Glenn Hammond, Peter Lichtner, and Chuan Lu. Subsurface multiphase flow and multicomponent reactive transport modeling using high-performance computing. *Journal of Physics: Conference Series*, 78(1):012025, 2007.

[26] J. A. Hartigan, Applied J. A. Hartigan M. A. Wong (1979) "A K-Means Clustering Algorithm", and M. A. Wong. A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[27] R. Hassan, A. Harris, N. Topham, and A. Efthymiou. Synthetic trace-driven simulation of cache memory. In *International Conference on Advanced Information Networking and Applications Workshop*, volume 1, pages 764–771, May 2007.

[28] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[29] David R. Kaeli. Issues in trace-driven simulation. In *Performance Evaluation of Computer and Communication Systems*, pages 224–244, London, UK, 1993. Springer-Verlag.

[30] George Karypis. Cluto a clustering toolkit. Technical report, University of Minnesota, Department of Computer Science, 2003.

[31] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Supercomputing*, 2001.

[32] Dimitri Komatitsch, Seiji Tsuboi, Chen Ji, and Jeroen Tromp. A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the earth simulator. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 4, Washington, DC, USA, 2003. IEEE Computer Society.

[33] S. Laha, J.H. Patel, and R.K. Lyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, 1988.

[34] M. Laurenzano, B. Simon, A. Snavely, and M. Gunn. Low cost trace-driven memory simulation using simpoint. In *Workshop on Binary Instrumentation and Applications*, 2005.

[35] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. *International Symposium on the Performance Analysis of Systems and Software*, March 2010.

[36] L.Carrington, N.Wolter, A.Snavely, and C.Lee. Applying an automated framework to produce accurate blind performance predictions of full-scale hpc applications. In *UGC*, 2004.

[37] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9:78 – 117, 1970.

[38] C. Mendes. Performance scalability prediction on multicomputers, 1997.

[39] Shuichi Mitarai, Masahiro Hirao, Tetsuya Matsumoto, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Compressed pattern matching for SEQUITUR. In *Data Compression Conference*, pages 469+, 2001.

[40] Richard C. Murphy and Peter M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Trans. Comput.*, 56(7):937–945, 2007.

[41] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, 2007.

[42] O. Y. Nickolayev, P. C. Roth, and D. A. Reed. Real-time statistical clustering for event trace reduction. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):144–159, Summer 1997.

[43] C. Olschanowsky, L. Carrington, and A. Snavely. A Tool for Characterizing and Succinctly Representing the Data Access Patterns of Applications. In *Submitted to IEEE International Symposium on Workload Characterization 2011*, 2011.

[44] Catherine Olschanowsky, Laura Carrington, Mustafa Tikir, Michael Laurenzano, Tajana Simunic Rosing, and Allan Snavely. Fine-grained energy consumption characterization and modeling. In *DOD High Performance Computing Modernization Program User Group Conference*, 2010.

[45] CM. Olschanowsky. Simulation guided performance modeling. Technical report, University of California at San Diego, Computer Science and Engineering Department, 2008.

[46] C.M. Olschanowsky, M.M. Tikir, L. Carrington, and A. Snavely. PSnAP: Accurate Synthetic Address Streams Through Memory Profiles. In *The 22nd Workshop on Languages and Compilers for Parallel Computing*, 2009.

[47] S. J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *The Journal of Computational Physics*, 117:1–19, 1995.

[48] G. Rote. Computing the minimum hausdorff distance between two point sets on a line under translation. In *Information Processing Letters*, 1991.

[49] P. Roth. Etrusca: Event trace reduction using statistical data clustering analysis. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[50] A. Samples. Mache: No-loss trace compaction. Technical report, University of California at Berkeley, 1988.

[51] T. Sherwood and et al. Automatically characterizing large scale program behavior. In *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 2002.

[52] Kevin Skadron, Margaret Martonosi, David I. August, Mark D. Hill, David J. Lilja, and Vijay S. Pai. Challenges in computer architecture evaluation. *Computer*, 36(8):30–36, 2003.

[53] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[54] Alan Jay Smith. Cache evaluation and the impact of workload choice. In *ISCA '85: Proceedings of the 12th annual international symposium on Computer architecture*, pages 64–73, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[55] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for application performance modeling and prediction. In *ACM/IEEE Conference on High Performance Networking and Computing*, 2002.

[56] M. Snir and J. Yu. On the theory of spatial and temporal locality. 2005.

[57] E.S. Sorenson and J.K. Flanagan. Evaluating synthetic trace models using locality surfaces. *IEEE International Workshop on Workload Characterization*, pages 23–33, Nov. 2002.

[58] David Sundaram-Stukel and Mary K. Vernon. Predictive analysis of a wavefront application using LogGP. *ACM SIGPLAN Notices*, 34(8):141–150, 1999.

[59] Robert E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Algorithmica*, 6(2):171–185, 1976.

[60] D. Thiebaut, J.L. Wolf, and H.S. Stone. Synthetic traces for trace-driven simulation of cache memories. *IEEE Transactions on Computers*, 41(4):388–410, 1992.

[61] M. Tikir, L. Carrington, A. Snavely, and E. Strohmaier. Genetic algorithm approach to modeling the performance of memory-bound codes. In *ACM/IEEE Conference on High Performance Networking and Computing*, 2007.

[62] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely. The pmac binary instrumentation library for powerpc. In *Workshop on Binary Instrumentation and Applications*, 2006.

[63] A. van Gemund and H. Lin. Scalability analysis of parallel finite element methods using performance simulation. In *EUROSIM*, 1995.

[64] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[65] Kapil Vaswani, Aditya V. Nori, and Trishul M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 351–362, New York, NY, USA, 2007. ACM.

[66] J. Weinberg and A. Snavely. Chameleon: A framework for observing, understanding, and imitating memory behavior. In *Workshop on State-of-the-Art in Scientific and Parallel Computing*, Trondheim, Norway, May 2008.

[67] Jonathan Weinberg. *The Chameleon Framework: Practical Solutions for Memory Behavior Analysis.* PhD thesis, Universithy of California at San Diego, La Jolla, CA, USA, 2008.

[68] W.S. Wong and R.J.T. Morris. Benchmark synthesis using the lru cache hit function. *IEEE Transactions on Computers*, 37(6):637–645, 1988.

[69] Xingfu Wu, Valerie Taylor, and Joseph Paris. A web-based prophesy automated performance modeling system. In *the International Conference on Web Technologies, Applications and Services (WTAS2006)*, 2006.

[70] Gengbin Zheng, Gagan Gupta, Eric Bohm, Isaac Dooley, and Laxmikant V. Kale. Simulating Large Scale Parallel Applications using Statistical Models for Sequential Execution Blocks. In *Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS 2010)*, number 10-15, Shanghai, China, December 2010.

[71] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 78, Santa Fe, New Mexico, April 2004.

[72] Yutao Zhong, Steven G. Dropsho, and Chen Ding. Miss rate prediction across all program inputs. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 79, Washington, DC, USA, 2003. IEEE Computer Society.