METHODOLOGIES AND TOOLS FOR COMPUTATION OFFLOADING ON HETEROGENOUS MULTICORES

A Thesis
Presented to
The Academic Faculty

by

Ashwini Bhagwat

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
College of Computing

Georgia Institute of Technology
August 2009

METHODOLOGIES AND TOOLS FOR COMPUTATION OFFLOADING ON HETEROGENOUS MULTICORES

Approved by:

Professor Santosh Pande, Advisor
College of Computing
Georgia Institute of Technology

Professor Sudhakar Yalamanchilli
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Nate Clark
College of Computing
Georgia Institute of Technology

Date Approved: 11 May 2009

To my family,

for their never ending support.

ACKNOWLEDGEMENTS

Firstly I would like to thank my advisor Dr. Santosh Pande for providing me with valuable insights and guidance for this Thesis. Throughout my Masters he has given me the opportunity to work on several interesting problems and has helped me grow in my research. I would like to acknowledge my colleagues Jaswanth Sreeram, Sunjae Park and David Zurow who have been involved in the development of certain parts of the infrastructure upon which this work is built. Jaswanth Sreeram has helped me tremendously in understanding several concepts and has guided me on many occassions. I would like to thank my colleague Farhana Aleen for her technical feedback and invaluable friendship. I would like to thank my committee members Dr. Sudhakar Yalamanchilli and Dr. Nate Clark for their valuable time and feedback. Many thanks to the Sony Toshiba IBM Consortium for funding my research.

I would not have come this far without the love and encouragement of my family and friends. I cannot thank them enough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Frequency scaling in traditional computing systems has hit the power wall and multicore computing is here to stay. Unlike homogeneous multicores that have uniform architecture and instruction set across cores, heterogenous multicores have differentially capable cores to provide optimal performance for specialized functionality. However, this heterogeneity also translates into difficult programming models, and extracting its potential is not trivial. The Cell Broadband Engine by the Sony Toshiba IBM(STI) consortium was amongst the first heterogenous multicore systems with a single Power Processing Unit(PPU) and 8 Synergistic Processor Units (SPUs). We address the issue of porting an existing sequential C/C++ codebase on to the Cell through compiler driven program analysis and profiling. Until parallel programming models evolve, the "interim" solution to performance involves speeding up legacy code by offloading computationally intense parts of a sequential thread to the co-processor; thus using it as an accelerator. Unique architectural characteristics of an accelerator makes this problem quite challenging. On the Cell, these characteristics include limited local store of the SPU, high latency of data transfer between PPU and SPU, lack of branch prediction unit, limited SIMDizability, expensive scalar code, etc. In particular, the designers of the Cell have opted for software controlled memory on its SPUs to reduce power consumption and to give programmers more control over the predictability of latency. The lack of a hardware cache on the SPU can create performance bottlenecks because any data that needs to be brought in to the SPU must be brought in using a DMA call. The need for supporting a software controlled cache is thus evident for irregular memory accesses on the SPU. For such a cache to result in improved performance, the amount of time spent in book-keeping and tracking at

run-time should be minimal. Traditional algorithms like LRU, when implemented in software incur overheads on every cache hit because appropriate data structures need to be updated. Such overheads are on off-critical path for traditional hardware cache but on the critical path for a software controlled cache. Thus there is a need for better management of "data movement" for the code that is offloaded on to the SPU.

This thesis addresses the "code partitioning" problem as well as the "data movement" problem. We present

GLIMPSES - a compiler driven profiling tool that analyzes existing C/C++ code for its suitability for porting to the Cell, and presents its results in an interactive visualizer.

Software Controlled Cache - an improved eviction policy that exploits information gleaned from memory traces generated through offline profiling. The trace is analyzed to provide guidance for a run-time state machine within the cache manager; resulting in reduced run-time overhead and better performance. The design tradeoffs and several pros and cons of this approach are brought forth as well. It is shown that with just about the right amount of runtime book-keeping and decision making, one can get to the difficult solution space of the right balance to achieve high performance.

CHAPTER I

INTRODUCTION

1.1 Multicore Computing

Traditionally, an increase in performance of computer systems has been achieved through an increase in processor frequency. The physical size of chips decreased, the number of transistors on chip increased. Clock speeds increased and heat dissipation rose to dangerous levels. In a single core, several techniques to improve performance have been introduced to gain performance, namely superscalar processing, instruction pipelining, out of order issue and branch prediction. However, once speeding up processor frequency had hit the power wall, computer architects decided to try a new approach to improve processor performance, and this led to the advent of multicore computing. By adding an additional processor, performance can be improved at modest clock speeds and lower heat dissipation. Thus in a multicore, performance boost is achieved without running at excessively high clock rates [19].

1.2 Heterogenous Multicores

The debate over whether cores in a multicore environment should be homogeneous or heterogeneous continues. In a homogenous environment, all cores are all exactly the same in terms of clock frequency, cache size and functionality. On the other hand, in a heterogeneous system, each core may have a different function, frequency and memory model. There clearly a tradeoff between processor complexity and customization. Homogeneous cores are easier to produce and program, because they have uniform instruction set and hardware across cores. However, some argue that this might not be the most efficient use of multicore technology. In a heterogenous environment, each

core can have a specific function, and run on its own special instruction set. The large centralized core can be generic and run the OS, while several cores can be specialized for different functionality like graphics, audio, floating point calculations etc. While this model is more complex, it is argued that its efficiency, power, and thermal benefits could outweigh its complexity. The Cell Broadband Engine(henceforth referred to as the Cell) was one of the first stable heterogenous multicore systems, with a single PPU and eight SPUs on a chip. While the work in this thesis is based on the Cell, it must be noted that the issues addressed here are likely to be encountered in any heterogenous multicore and the tools and approach described are generic enough to be extended to other similar environments. [19]

1.3 Programming Model

In May 2007, Intel fellow Shekhar Borkar stated that "The software has to also start following Moore's Law, software has to double the amount of parallelism that it can support every two years." Since the number of cores in a processor is set to double every 18 months, it makes sense that the software running on these cores takes this into account. Ultimately, programmers need to learn how to write parallel programs that can be split up and run concurrently on multiple cores instead of trying to exploit single-core hardware to increase parallelism of sequential programs [19]. The Cell is a high performance, heterogeneous, parallel architecture that is well suited for a diverse range of workloads ranging from scientific applications to digital home entertainment applications. It has been reported to be capable of around 230 GFLOPS at 3.2 GHz for single-precision floating point operations. However, achieving this computational throughput requires the programmer to understand both his application and the architecture thoroughly. There are two possible ways to use a multicore. One way could be to run a multithreaded program on multiple cores to gain speedup due to parallelization. However this involves complex programming models. The other way

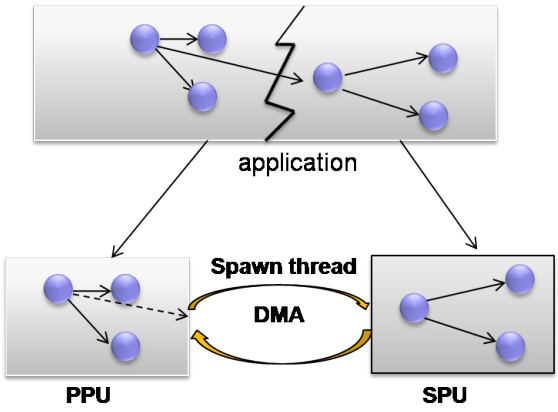

Figure 1: Program Partitioning

is to use the cores as accelerators by pushing offloading compute intensive code that optimally uses the capability of the multicores. Until parallel programming models evolve this approach is more likely to be used and is the model of our focus.

1.4 Challenges

Prototyping large legacy sequential codebases for porting on to the SPEs of the Cell involves several challenges. The first problem is the one of selecting the optimal partition or set of functions to execute on the PPE and the SPEs. We refer to this issue as /emphCode Partitioning. An abstract view of partitioning a program on the function call graph is depicted in Figure 1.

Apart from code, data that is shared between the PPE and the SPEs also needs to be synchronized automatically. This is the /emphData Movement issue faced in any distributed-memory architecture and is also not trivial to automate. The optimal partition to be ported on to the SPU must be identified keeping in mind the architectural constraints of the Cell and the SPU. These constraints are limited local store of 256 KB, lack of branch prediction unit on the SPU, and the fact that it is geared towards vectorizable rather than scalar code. The set of upward exposed references must be identified and managed appropriately. Function arguments and

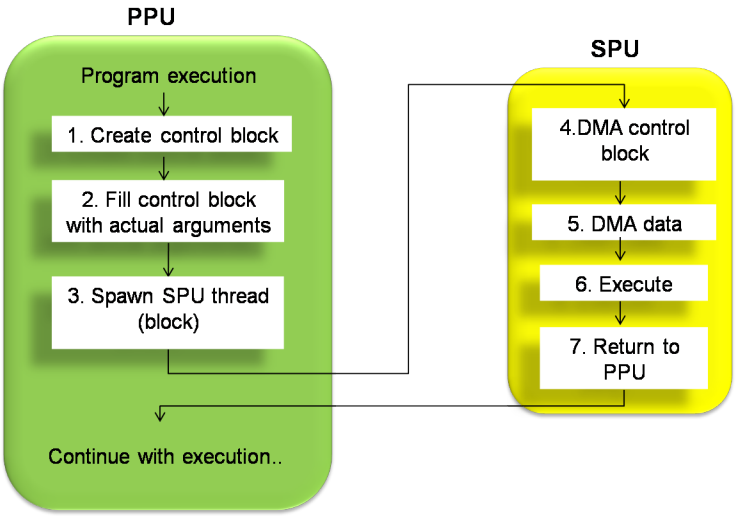

Figure 2: PPU SPU Communication

returns must be handled using the control block DMA mechanism. This is depicted in Figure 2

Apart from this, any remote data references must also be managed via DMA transfers. The data on SPU needs to be 128 byte aligned as well. Thus it is evident that we need to provide a programmer with tools to understand program behavior, and quickly construct candidate partitions for the SPE obeying its architectural constraints. There is also a need to evaluate and quantify the suitability of a set of functions to be ported to the SPEs with respect to performance. Once the functions are ported on to the SPU, there is a need to manage data in a more programmer friendly manner. The rationale for the architecture of the Cell in terms of power efficiency is can be found in [17].

In this thesis, we describe a two-pronged approach to address these problems:

- *Porting a legacy codebase* - A compiler driven profiling tool called GLIMPSES that enables programmers to understand the static and dynamic behavior of their programs, quickly construct candidate partitions for the SPEs and evaluate and quantify the suitability of these partitions for execution on the SPEs. This

is meant to solve the code partitioning problem and is described in Chapter 3

- *Managing a ported codebase* - A Software Controlled Cache for seamlessly managing data transfer between SPUs and PPUs. This is meant to solve the data movement issue and is described in Chapter 4.

An overview of the Cell provided in Chapter 2.

CHAPTER II

THE CELL BROADBAND ENGINE

The Cell Broadband Engine processor by the STI consortium is a novel heterogeneous multicore architecture. It consists of a single main core called the Power Processing Engine(PPE) and eight Synergistic Processor Engines(SPE)s. The PPE acts as the controller that runs the Operating System(OS) and dispatches and manages jobs on the SPE. The SPE is suitable for compute intensive tasks, offers very high processing power and supports SIMD parallelism. The Cell architecture is designed to be well-suited for a wide variety of programming models and allows for partitioning of work between the PPE and the eight SPEs. Figure 3 shows the architecture of the Cell Processor.

2.1 Power Processor Element (PPE)

The PPE is a Power Architecture based core that acts as the controller for the eight SPEs. It has control over the SPEs and can start, stop, interrupt and schedule processes running on the SPEs. Unlike SPEs, the PPE can read and write the main memory and the local memories of SPEs through the standard load/store instructions. The PPE works with conventional operating systems that can run on 64-bit PowerPC processors, while the SPEs are designed for vectorized floating point code execution. The PPE contains a 32 KB instruction and 32 KB data Level 1 cache and a 512 KB Level 2 cache [2].

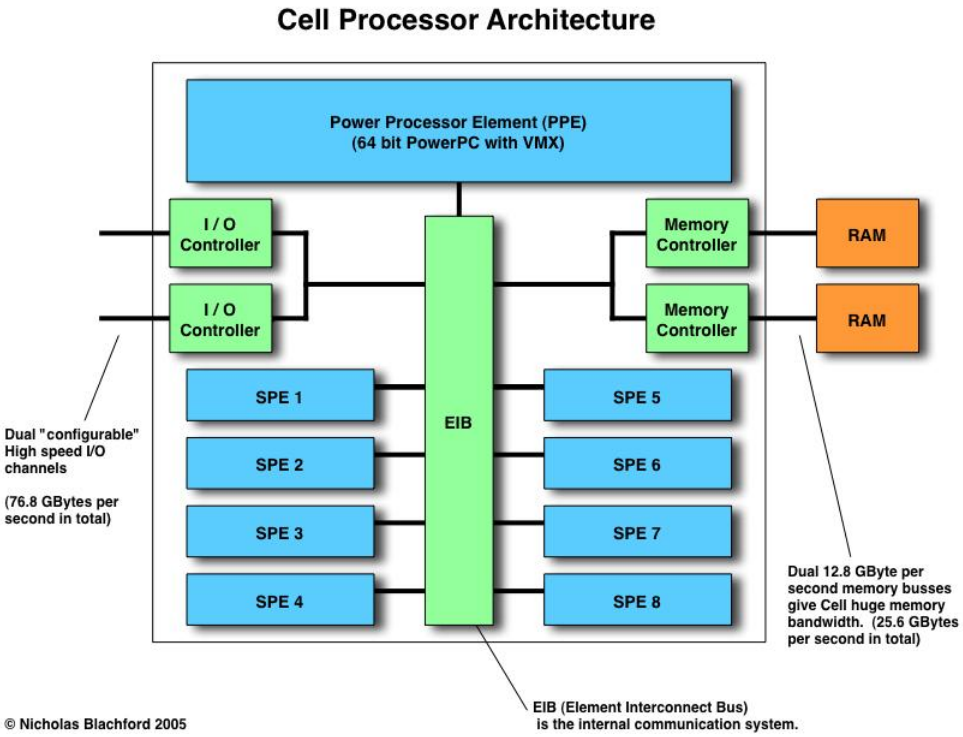

Figure 3: Cell Processor Architecture [15]

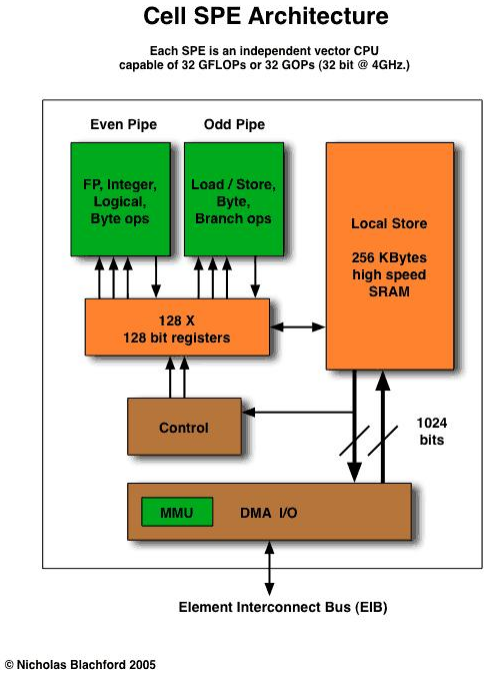

Figure 4: Cell SPE Architecture [15]

2.2 Synergistic Processing Element (SPE)

The SPE is meant to handle the computational workload of the program. Each SPE is composed of a Synergistic Processing Unit(SPU) and a Memory Flow Controller(MFC). The MFC consists of DMA, MMU and Bus Interface. Each SPE contains a 256 KB embedded SRAM for instruction and data called "Local Storage", which is visible to the PPE and can be addressed directly by software. Each SPE can support up to 4 GB of local store memory. The local store does not operate like a conventional CPU cache since it is neither transparent to software nor does it contain hardware structures that predict which data to load. The SPU cannot directly access system memory; the 64-bit virtual memory addresses formed by the SPU must be passed from the SPU to the SPE memory flow controller (MFC) to set up a DMA operation within the system address space. Although the SPEs have Turing complete architectures, they are not fully autonomous and require initiation from the PPE.
Both the PPE and SPE are RISC architectures with a fixed-width 32-bit instruction format. The PPE contains a 64-bit general purpose register set (GPR), a 64-bit floating point register set (FPR) and a 128-bit Altivec register set. The SPE contains 128-bit registers only. These can be used for scalar data types ranging from 8-bits to 128-bits or for SIMD computations on a variety of integer and floating point formats.

2.3 Element Interconnect Bus (EIB)

The Cell marries the SPEs and the PPEs via the Element Interconnect Bus(EIB). The EIB gives access, via fully cache coherent DMA to main memory and external storage devices. To make the best of EIB and to overlap computation and data transfer, each of the nine processing elements (PPE and SPEs) is equipped with a DMA engine. Since the SPE's load/store instructions can only access its own local memory, each SPE entirely depends on DMAs to transfer data to and from the main memory and other SPEs' local memories. A DMA operation can transfer either a

single block area of size up to 16KB or a list of 2 to 2048 such blocks. One of the major design decisions in the architecture of the Cell is the use of DMAs as a central means of intra-chip data transfer, with a view to enabling maximal asynchrony and concurrency in data processing inside the chip.

CHAPTER III

THE GLIMPSES TOOLKIT

3.1 Introduction

GLIMPSES is an acronym for Global Interprocedural Memory and Partition Estimator for SPEs. It is a tool for analyzing legacy sequential codebases for the purpose of porting them to the SPEs of the Cell[18]. This is a challenging task because of the novel architectural attributes of the processor and the amount of parallelism available. GLIMPSES is a compiler-driven profiling and visualization framework that enables Cell programmers to quickly evaluate the static and dynamic behavior of a program to determine its suitability for execution on the SPEs. With this tool, programmers can view call graphs, function characteristics such as stack and heap usage and dynamic memory reference patterns in an interactive visualizer. We describe the overall approach used, information that the tool can produce, why this information is useful for a Cell programmer and how they are represented in the tool.

The first version of GLIMPSES was developed by Jaswanth Sreeram. I have worked on the tool and incrementally added features after the 0.87 release. The parts that I am specifically responsible for have been highlighted. However, for the sake of completeness, all its features have been described, with appropriate citations where applicable.

The tool is licensed under an open-source BSD license. It is hosted at `http://sti.cc.gatech.edu/software.html` and is available for download at `http://glimpses.sourceforge.net`.

3.2 Related Work

There exist several tools for profiling applications. Some of these are elaborated below.

- *gprof* - The GNU profiler gprof [7] is a useful tool for measuring the performance of a program–it records the number of calls to each function and the amount of time spent there, on a per-function basis. Functions which consume a large fraction of the run-time can be identified from the output of gprof. The idea is that efforts to speed up a program can concentrate first on those functions which dominate the total run-time.

- *Valgrind* - Valgrind [13] is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile programs in detail. Valgrind can also be used to build new tools. The Valgrind distribution currently includes six production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache profiler, and a heap profiler.

- *Pin* - Pin [10] is a tool for the dynamic instrumentation of programs. Pin does not instrument an executable statically by rewriting it, but rather adds the code dynamically while the executable is running. Pin includes the source code for a large number of example instrumentation tools like basic block profilers, cache simulators, instruction trace generators, etc. It is easy to derive new tools using the examples as a template.

- *Eclipse PTP* - The Eclipse Parallel Tools Platform [12] provides a highly integrated environment for parallel application development.The aim is to provide a standard, portable parallel IDE that supports a wide range of parallel architectures and runtime systems; a scalable parallel debugger; support for the

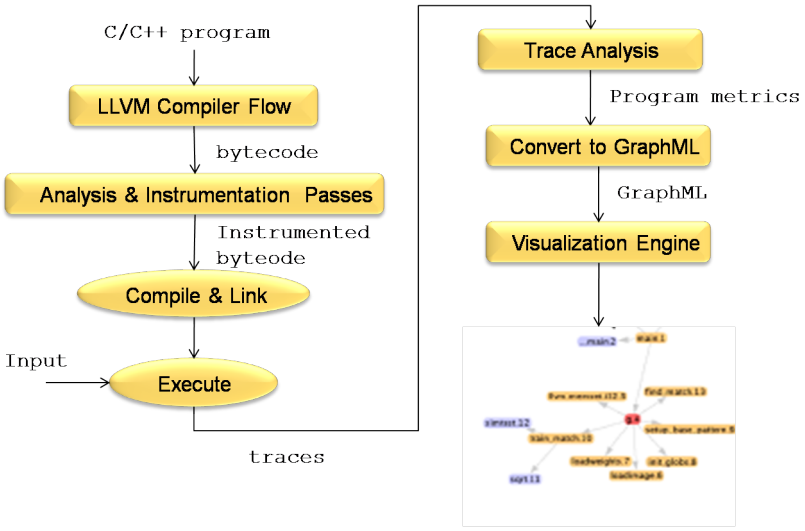

Figure 5: GLIMPSES Tool flow

integration of a wide range of parallel tools; and an environment that simplifies the end-user interaction with parallel systems.

- *Intel VTune* [6] - The Intel VTune Performance Analyser is a commercial application for software performance analysis for x86 based machines [14].

3.3 Approach

GLIMPSES consists of several components: a compile-time instrumentation pass, post-execution analysis passes and a graphical visualizer[18]. The process of producing viewable profiling information for a program is shown in Figure 5.

A sequential C/C++ program is compiled with the base compiler which is LLVM [8] in this case. LLVM (Low Level Virtual Machine) is an optimizing compiler that enables several advanced compiler analysis and optimizations. It is noted that all the analysis and instrumentation performed in the LLVM compiler for producing traces can also be added to other compilers i.e. the tool is not dependent on compilation features present only in the LLVM compiler. LLVM generates a machine independent bytecode, which its the intermediate representation(IR) on which the instrumentation

pass is run. This instrumentation pass takes as input the original bytecode and produces bytecode in which all the events of interest are instrumented. These are:

- Function entry and exit points
- Loads and Stores
- Dynamic Memory Allocation instructions

The instrumented bytecode is then compiled and linked. On running the new executable with test input, a trace of the events of interest is generated. This trace is used by all analysis passes to extract information that is useful to the programmer. The extracted data is translated into GraphML [5] format which is understood by the prefuse [11] framework and rendered in the Visualization framework. The feature set of GLIMPSES is elaborated in Section 3.4. A sample GraphML file for the Static Call Graph of a small test application is shown in Appendix C. Each node corresponds to a single function in the graph, which has various function metrics associated with it. The edges represent the function calls.

3.4 Features and Visualization Framework

The features of GLIMPSES can be broadly classified into three categories: those that aid the programmer in understanding the program as a whole, those that aid in understanding individual functions and those which help in making code partititioning decisions. This is depicted in Figure 6. Support for features Dynamic Call Graph, Memory Locality Behavior and Function Metrics- Code Size, Stack Size, Autovectorizable loops and Branch Density was present in GLIMPSES 0.87. I have been involved in four releases (GLIMPSES 0.91 - 0.94) and the incremental feature addition that were involved in these releases. These are highlighted in green in the Figure and elaborated in sections that follow. However all features have been described below for the

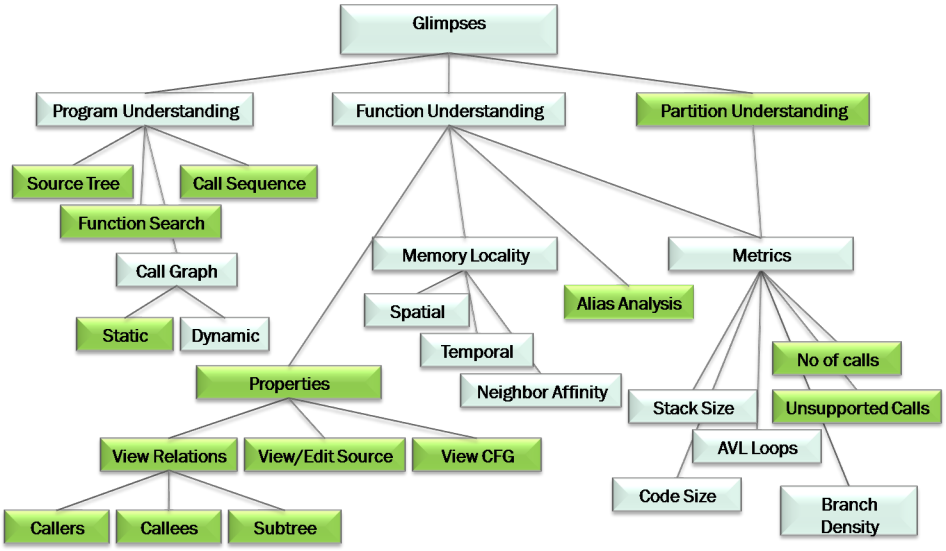

Figure 6: GLIMPSES Features

sake of completeness. The use cases for GLIMPSES are depicted with screenshots in Appendix B

3.4.1 Evaluating the program

3.4.1.1 Function Call Graphs

The function call graph gives the programmer an idea of the overall structure and caller-callee relationships in the program. We present to the programmer, both the Static and the Dynamic Call Graph. The Static Call Graph consists of all functions that are defined in the workload, one node per function. The nodes are colored green or red to distinguish whether it was called during the profile run or not. The Dynamic Call Graph consists of only those functions that were called during the profile-run, one node for each invocation or instance of the called function. They are time-stamped to indicate the order in which they were called.

3.4.1.2 Function Stepping

To allow for overlaying of the dynamic program run on the Static Call Graph, there is a feature to allow the programmer to play the actual call sequence that occurred

during the profile run through function stepping.

3.4.1.3 Source Tree

The programmer can view the Source Tree of the program on the left pane of the visualization framework, which highlights the appropriate file as and when a certain function is selected on the Static Call Graph. This feature is useful for a programmer to go back and edit the file based on the feedback provided by the tool.

3.4.1.4 Function Search

When workloads are very large, it can be difficult to locate specific functions and observe their features. Therefore, a search feature is implemented, which will highlight the desired function on the Source Tree as well as the Static Call Graph if it is present.

3.4.2 Evaluating a function

When the programmer hovers the mouse pointer over any function i.e. a node in the Static or Dynamic Call Graph, its properties are updated on the function properties panel on the right.

3.4.2.1 Memory Locality Behavior

[18] Due to the limited size of the local store on the SPEs, understanding the memory reference behavior of the program can help the programmer in laying out the data such that it can be efficiently moved between an SPE local store and local stores of other SPEs or main memory. Ideally, one would like functions executing on the SPE to reference a small working set of data, very frequently. If that is not the case then re-layout of data can reduce both the amount of data that is unnecessarily fetched and the amount of data that has to be flushed to main memory in order to bring in new data into the local store. GLIMPSES can produce three types of locality information from the dynamic memory reference trace that is produced during execution of the program. The three types of locality reported are:

Spatial Locality - This property is a measure of the number of loads to addresses in a spatial window. It indicates the probability that if a particular word in memory is brought into the local store, then future loads access the memory line that word falls in. Thus the higher the spatial locality, the more spatially clustered memory references are.

Temporal Locality - This property is a measure of the number of loads to the same address in a time window. Given a load instruction executing at a particular point in the program, this measure tells us, how many future loads will access this same memory address. The higher the temporal locality, the more frequently the address is accessed, and hence the longer it should be kept around in the local store.

Neighbor Affinity - This is a property that combines both spatial and temporal locality. Specifically, neighbor affinity is the probability that loads that occur during a window in time, also access memory addresses in a window in memory. It is computed as follows: Given a load in the trace that is accessing a particular memory address X and given a window of size d in the trace starting at this load, the neighbor affinity of this load is equal to the number of loads in that window that access memory addresses that are not more than a memory line away from X.

3.4.2.2 Function metrics

The function metrics that are currently being displayed are: *Code size*: The Code Cize is simply the number of bytes required to store the code for this function definition on the SPU. Let this be S1.

Used Code Size: When a set of functions are selected, the Used Code Size is the total code size of only the functions that were called during the profile run. Let this be S2.

Code Utilization Ratio: This metric is an indication of the amount of code that was actually exercised. Thus it can be calculated as S1 / S2.

Stack Size: The Stack Size is the number of bytes of stack space used in the function

invocation during the profile run.

Heap Size: The Heap Size is the number of bytes of heap space that was used in that function invocation during the profile run.

Branch Density: The Branch Density of a function is calculated as the ratio of the number of branch instructions in the function to the total number of instructions.

Number of Autovectorizable loops: This is simply the number of auto-vectorizable loops found by the gcc compiler in the given function.

Number of calls: This number is the number of times this function was called during the profile run.

Number of Unsupported Function Calls: This is the number of external unsupported function calls that are in this function definition.

The tool allows the programmer to select more than one function. As new functions are added or removed under the selection, the function properties are appropriately updated to display the effective aggregate value. This can help a programmer understand the effect of choosing the set of functions to be ported on the SPE.

3.4.2.3 Utilities

There are several other utility features provided in GLIMPSES to improve the comprehension of specific functions. These are listed below. When the user right clicks his mouse on any node in the static call graph, the tool pulls up a context menu where the user may choose to do one of the following

1. View and Edit source code of the function: GLIMPSES opens up the code for the function in the emacs editor. This can be very useful for editing source code as the programmer views its properties.

2. View Control Flow Graph(CFG) for the function: The CFG is generated with support from LLVM and gives a very good picture of the branching and loop

structure of the function.

3. View the callers of the function: Upon choosing this option, all functions that could potentially call the selected function are highlighted in yellow.

4. View the callees of the function: Upon choosing this option, all functions that the selected function could potentially call are highlighted in yellow.

5. View all functions reachable from the function: Upon choosing this option, all functions that can be reached through calls from this function are highlighted in yellow.

6. Select all functions reachable from the function: Upon choosing this option, all functions that can be reached though calls from this function are highlighted in deep green and red (depending on whether they were called or not during the profile run respectively) and their function metrics are aggregated and displayed appropriately.

3.4.3 Evaluating a partition

A key decision a Cell programmer has to make is the partitioning of code between the SPEs and the PPE. To help the programmer in making this decision, GLIMPSES has features to analyze the whole program, or observe specific functions as described. It also allows the the programmer to select a set of functions for which he may view aggregate metrics. The code utilization ratio is also an indication of the amount of code that might be actually exercised when pushed on to the SPU. The tool can be used to filter out bad selections by allowing the user to specify thresholds on each function metric, weeding out bad selections, and reducing the problem space. Thus a user could specify that he wants only functions with branch density greater than X, or code size less than Y or a combination to be displayed.

Table 1: GLIMPSES Release History

Feature	0.87	**0.91**	**0.92**	**0.93**	**0.94**
Dynamic Call Graph	√	√	√	√	√
Memory Map	√	√	√	√	√
Function Properties	√	√	√	√	√
Alias Analysis			√	√	√
Static Call Graph			√	√	√
Function Stepping				√	√
Control Flow Graph				√	√
Function Search				√	√
Source Tree				√	√
Callers and Callees				√	√
Subtree				√	√
View/Edit Source				√	√
Unsupported function warnings					√
PowerPC Estimates					√

3.5 Conclusion

3.5.1 Contribution

GLIMPSES is generic tool that allows one interactively analyze and port a legacy sequential codebase on to the Cell SPU. The architecture of the tool is such that it can easily be configured to extend to any heterogenous multicore. It is modular enough to allow for variation of the base compiler, or change in metrics to be observed. The current download size of GLIMPSES from sourceforge is 13.8 MB. This includes LLVM and prefuse for the Visualizer. When built, its executable size is 200.6 MB. It can handle very large traces. For the mpeg2dec application, the trace size was 630.2 MB. I have been the developer for the GLIMPSES toolkit through four releases during August 07 - September 08. These involve incremental feature addition though releases 0.91,0.92, 0.93 an 0.94. The release history with feature sets since GLIMPSES 0.91 has been shown in Table 1. The 0.91 release included several User Interface changes, but did not add any new features. There have been 190 downloads over the past year from sourceforge.

3.5.2 Future Directions

GLIMPSES garnered a lot of interest during the STI Cell Workshop at Georgia Tech in July 2008. Terrasoft Solutions was interested in a distribution of GLIMPSES for its version of Yellow Dog Linux. IBM was interested in exploring GLIMPSES as a plug-in for Eclipse Parallel Tools Platform. These are being investigated. There are several features could be built into the tool. These are listed below.

- Interprocedural Alias Analysis information for providing for refining partition sets. Such analysis can expose a lot of non-obvious function dependence information. The aliasing results can be used to estimate a new metric which indicates the coupling between two functions; Higher the probability of aliasing, the more tightly coupled they are, and this would mean that placing these two functions on different SPUs would be a bad idea as it would introduce more synchronization overhead. Alias analysis information can also be used for data pinning and prefetching. With precise function dependence information, it may be possible for the compiler to outline regions of memory that should be prefetched before they are actually needed. Macroscopic data structure analysis can be leveraged to provide smart heap memory allocation on the SPEs that have been found to be very effective in improving performance in pointer-intensive workloads. Improved and intuitive visualization schemes could be investigated for the representation of memory access patterns, alias sets and partitions.

- Integration into the IBM Eclipse Toolkit as a plugin.

- GLIMPSES currently performs estimation on the x86 because of the limitations of the LLVM compiler that has limited support for PowerPC on Linux. However there are architectural differences between x86(which is a CISC processor) and Powerpc(which is a RISC processor) and further differences between the

instruction sets on the PowerPC and SPU. GLIMPSES could be installed on the PowerPC with a different base compiler to provide for more accurate results.

- GLIMPSES could be extended into a more generic tool for the analysis of sequential code for any multicore system through the use of machine description files provided as metadata, thus providing different perspectives for different multicores and not limiting itself to the Cell. This can greatly improve its reach and utility in the future and could be used to solve the computational offloading problem for GPUs and DSP co-processors as well.

CHAPTER IV

SOFTWARE CONTROLLED CACHE

4.1 Motivation

The Cell emphasizes efficiency, prioritizes bandwidth over latency, and favors peak computational throughput over simplicity of program code. [2] Though most of the "horsepower" of the Cell comes from the SPEs, the use of DMA as a method of data transfer and the limited local memory footprint of each SPE pose a major challenge to software developers who wish to make the most of its potential. This demands careful hand-tuning of programs to extract maximal performance from the Cell. Programmers have to manage the SPE execution including creation of threads, managing data transfers and aligning the data on local stores among other issues. The local stores on the SPEs are limited in size (256 KB) and make it mandatory for a programmer to manage the local memory space efficiently and manually. This can severely impact programmer productivity. DMA transfers and non-local accesses in SPEs can create a huge performance overhead that can lead to unacceptably slow execution on SPEs. The Cell is widely regarded as a challenging environment for software development. IBM provides a comprehensive Linux-based Cell development platform to assist developers in confronting these challenges through the IBM Cell SDK. The difficult programming model of the cell continues to be the main bottleneck to its adoption. The need for a Software Cache is evident as it can improve application performance by providing a mechanism to look up data in local storage before attempting a DMA transfer. This could save program several cycles. There is also a need to provide more transparent management of this data movement to simplify programming and pave

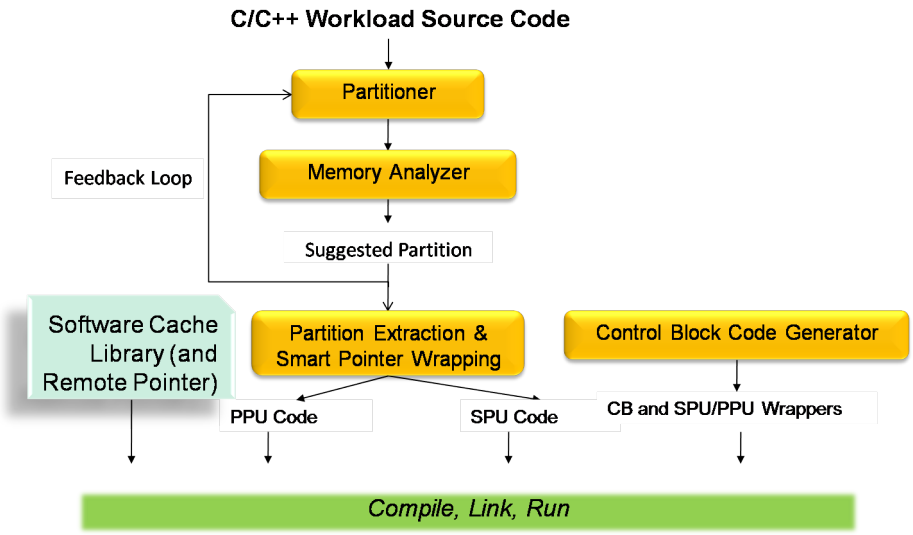

Figure 7: The Software Cache ToolChain

the way for wider adoption of the Cell. A Software Cache Toolchain has been developed at our research group. It is a comprehensive solution that not only provides a caching mechanism to improve application performance; but also a system that enhances programmer productivity and code quality. Since it is the infrastructure upon which my work is built, an overview of the system is elaborated in Section 4.2. Sections that follow go into the details of my work.

4.2 The Software Cache Toolchain

The Software Cache Toolchain proposes a solution of automatically managing the SPE local store and also provides a seamless interface for the management of SPE - PPE interactions for data and code transfers. The solution consists of a compiler-based profiler and analyzer, a single-source compilation system consisting of source-to-source code generator and a runtime system on SPEs that manages local SPE store and data spaces. The solution is based on profile based analysis that allows understanding dynamic behavior of the code. A compilation phase called "partitioner" first identifies the call sites that can be offloaded to the SPE from a C/C++ source level program and

then generates the necessary code which manages the PPE/SPE interactions. The code generator outputs PPE code and SPE code at C/C++ source level. The software cache is a runtime system that works closely in conjunction with the compiler analysis and hints generated there from. The phase which generates these hints is called the "memory analyzer". The generated PPE and SPE code can then be compiled using the native PS3 gcc compilers. The SPE code is transformed to use "smart pointers" wherever there exist remote pointer dereferences. The smart pointers are a template based solution that transparently invoke the routines in the software cache library to first check if the memory reference exists in the local store, and if it does not, DMA the data into the local store and access it. For the purpose of understanding the overall motivation of this work, the Paritioner and Memory Analyzer are briefly described in Section 4.2.1.

4.2.1 Partitioner and Memory Analyzer

The Partitioner analyzes the source code of a program and generates suitable partitions that can be run on the Cell, such that considerable computation is offloaded to the SPUs. A partition is defined as a subtree of the dynamic callgraph, such that there is a single entry of control flow into and out of the partition (through the root of the partition subtree).

The Partitioner analyzes both the static and dynamic characteristics of the program sources and attempts to maximize the amount of execution time spent inside the partition. It considers various characteristics of each callgraph node in the light of the limitations of the SPU. These are noted below.

- *The SPU does not have a hardware branch predictor* : This means that code with heavy branches will suffer slowdown when run on the SPU. Therefore, the partitioner checks the dynamic conditional and unconditional branch ratio and will reject those with too high ratios.

- *The SPU has a tight storage constraint* : The SPU local store has 256K available for text, stack, and heap. The software controlled cache that will reside on the SPU also needs to be accounted for as it will dynamically allocate memory for the cache. The partitioner uses the profiling runs and determines the storage required.

- *The SPU is a 128 bit SIMD processor* : Scalar code can lead to frequent loads, stores and vector shuffles. The partitioner identifies code that is sure to produce this type of behavior and requires that they are infrequent.

- *The Software cache adds to the slowdown a partition experiences* : The overhead caused by the software cache also needs to be low to allow for improved performance

The Memory Analyzer works in conjunction with the partitioner to predict the possible memory allocation and usage for a proposed partition set. It analyzes memory traces to determine appropriate design parameters for the Software Cache; and simulates the result for approximate performance statistics. Finally the analyzer feeds back to the partitioner the estimated slowdown due to the software cache (due to inherent maintenance overhead and DMA transfer stalls for misses) to improve partitioning. Through this continuous feedback, it refines the partitions, possibly rejecting some for performance or space constraints.

The Partitioner and Memory Analyzer are treated as a black box for the purpose of this thesis and is not central to my work. However, the Software Cache Library and Smart Pointer APIs are used extensively and are described in Section 4.2.2 and 4.2.3. This library is developed by Sunjae Park.

4.2.2 Software Cache Library

The Software Cache is an object based line backed software cache that lies between the address spaces of the PPU and SPU. As with any subroutine, functions offloaded

on the SPU partition may require certain amount of external data for correct execution. Simple data can be passed through arguments and explicit DMA, but this can become complicated with code when there is heavy SPU-PPU interaction through the use of pointers. The Software Cache accepts PPU addresses and will bring in data automatically as they are accessed. Coupled with the remote_ptr smart pointer, the software cache can provide near transparent access from the SPU to the PPU address space. The smart pointer is explained in Section 4.2.3.

In addition to the DMA overhead, the Software Cache executes as part of the SPU partition and adds a certain amount of overhead. The Software Cache API acts as a low-level interface to the Software Cache. In addition, the Software Cache will automatically handle issues that occur with alignment. A user just needs a pointer to the PPU address desired, and the size of the object. The use of the remote_ptr API, which acts as a wrapper around the Software Cache API, is encouraged. The current implementation of the toolchain assumes that objects brought into the SPU via this Software Cache API does not exceed the size of a single line.

4.2.3 Smart Pointers

The remote_ptr acts as a wrapper around the Software Cache API and is the recommended way to interact with the software cache. It wraps a remote pointer reference in a declaration of a template class which in-turn calls Software Cache routines. This transparent management of data transfer details has various benefits:

- Legacy code can be easily ported to the Cell as the Memory Analyzer and Partitioner analyze code to suggest optimal partitions after analysis in the form of a Single Source Compilation system.

- The programmer is freed from the task of finding suitable partitions and also managing the cache, greatly improving his productivity.

- A caching mechanism for the SPU improves application performance by saving DMA transfer cycles in cases where data has already been brought into the local store.

- The Smart Pointer template based solution abstracts away the details of data transfer and vastly improves code quality.

A simple example of a program ported to the Cell through the use of the Software Cache Toolchain as well as a sample Makefile has been provided in Appendix D.

4.3 Cache Eviction

The main contribution of my work is the exploration and implementation of a trace driven eviction policy built into the Software Cache Infrastructure described in 4.2.2. To fully understand the mechanism of cache line eviction when a remote pointer is dereferenced on the SPU, we first look at cache eviction in general in Section 4.3.1. Section 4.3.2 describes the Least Recenty Used (LRU) policy for eviction as it is one of the most widely implemented eviction policies in hardware and software. The limitations of LRU are brought forth as well. To be able to compare the newly implemented trace driven eviction policy it is benchmarked against an LRU implementation in the Performance Evaluation. For the purpose of this thesis, the software controlled cache is always assumed to be fully associative cache with a configurable line size and number of lines.

4.3.1 Eviction Policy

Figure 8 shows the a cache eviction mechanism for a fully associative cache in general. When an remote address is dereferenced, the cache manager checks if it exists locally. If so, it simply retrieves it from there. If it is not found locally, it needs to bring it in. For this, the cache manager first looks up if there is free space in the cache to bring in the required line. If there is a free line, it can bring it in immediately. If not, a line

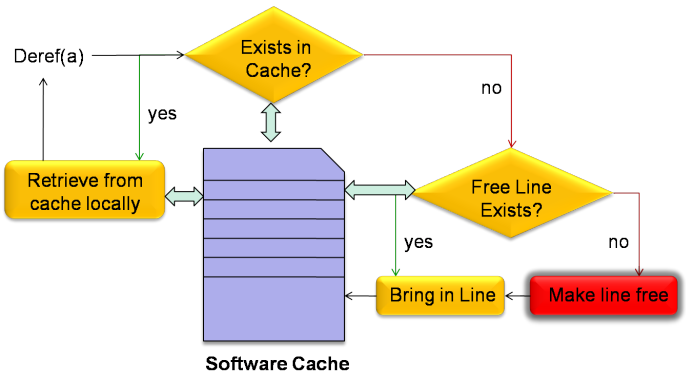

Figure 8: Cache Eviction Policy

is freed by evicting something [and writing back if it is dirty] to make space. What is evicted depends on the replacement policy used.

4.3.2 LRU Eviction Policy

LRU works on the principle that references that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory, it is rather expensive to implement in practice. There are a few implementation methods for this algorithm that try to reduce the cost yet keep as much of the performance as possible. The most expensive method is the linked list method, which uses a linked list containing all the objects in memory [9]. At the back of this list is the least recently used page, and at the front is the most recently used references. This is costly because items in the list will have to be moved about every memory reference. There is considerable overhead even when there is a cache hit, because the line needs to be pulled out of the middle of the list and pushed to the back. This overhead can be somewhat minimized on a miss, by parallely maintaining a hash map of references as well, and looking up this map to check if the address exists before searching the list. So a linear search on the list will only be required in the case of a hit. However this is a time-space tradeoff. While there can be more efficient ways to do it with hardware support, this

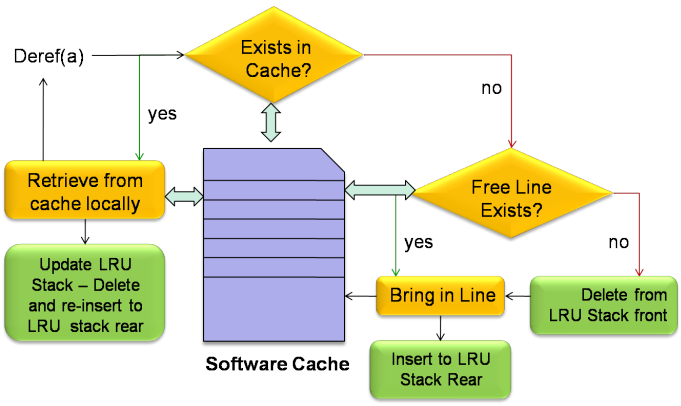

Figure 9: LRU Cache Eviction

overhead cannot be sufficiently reduced in a software implementation. Apart from the implementation costs discussed, LRU also tends to degenerate under certain reference patterns. For instance, with a fully associative cache with N lines, a program with a loop over N+1 references will continuously suffer cache misses. LRU only works well when a program adheres to locality of reference, and when past references are a good indication of future references. In such a set up one can hope that once the cache is filled, it will suffer fewer misses.

The overall flow in of cache replacement (LRU) in the Software Cache Toolchain is depicted in Figure 9. An LRU List is maintained. When an address is dereferenced, the cache manager searches through this list to check if it exists locally. If so, it updates the list by removing this address from its position in the list and re-inserting it to the rear. If the address does not exist locally, it is brought into the cache by evicting the line from the front of the stack. Thus there is considerable book-keeping to be done during run time. Improvements to LRU with the use of additional in-memory data structures turned out to be infeasible because of the memory restrictions of the Cell SPU.

4.3.3 Opportunities for optimization

As discussed, traditional replacement policies like LRU incur a considerable run time overhead and this must be minimized if we want to extract performance, while still retaining the several benefits of a software controlled cache. As per Belady's Algorithm the most efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. Since it is generally impossible to predict ow far in the future information will be needed, this is not implementable in practice. However, in our case, we can leverage memory trace information from profiling to predict possible future references in a long running program to improve performance. This is an advantage that we get in a software controlled cache as it is highly reconfigurable and can be hand-tuned. We must keep in mind though, that while optimizing eviction for performance, the overhead on each reference needs to be reduced. Again, this flexibility comes in a software controlled environment where we can specify exactly what we want to track and how.

4.4 Related Work

The IBM Cell SDK 3.0 provides a software cache as a library which can be used by application programmers. It provides for two modes, a synchronous mode and an asynchronous mode. The software cache can be configured based on Associativity, Access mode (Read-only or Read-Write), Cache line size, Number of lines and Data type. The synchronous mode provides the programmer with a set of functions to access data simply by using the effective address. The software cache library performs the data transfer between the Local Store(LS) of the SPU and the main memory transparently to the programmer and manages the data that is already in the LS. The asynchronous interface enables the programmer to hide the memory access latency by overlapping data transfer and computation. This mode provides a more efficient means of accessing the LS compared to the safe mode. The software cache provides

Table 2: Call Stacks and Instances

Benchmark	Total Function Calls	Total Unique Call Stacks
grep	1262	256
diff	4307	2623
mpeg2dec	635921	193
awk	9780	113
md5sum	52	24

functions to map effective addresses to the LS addresses. The programmer should use those LS addresses later to access the data, unlike in safe mode where the effective addresses are used. There is also a provision to define multiple caches, each configured differently to suit the needs of the programmer. [16] However it must be noted that while the IBM SDK provides for software cache routines, the instantiation of this cache and management of data in and out of it need to be explicitly handled by the application programmer. Our toolchain provides the Smart pointer wrapper that is a layer on top of the cache which greatly simplifies its management. Further, it is a comprehensive single source compilation toolchain that takes in regular C programs to convert it into Cell compliant code.

4.5 Approach

4.5.1 Concept

The central idea is to exploit memory trace information to be able to make more informed decisions for eviction during run time. By predicting the references that might be needed in the future, we could reduce the miss penalty that would suffer the DMA overhead. Also, since LRU requires active book keeping and tracking, we need to try to reduce this by tracking less frequently and making the eviction more light weight. Program behavior is often not a random phenomenon. Memory references are normally propagate through the stack of function calls.

Table 2 shows the number of call stacks as against the total number of function

calls in a program. We can see that there are a fixed number of function call stacks that are repeatedly seen as the program enters and exits the same functions. For such long running programs that consist of several loops, we can benefit from aggregating behavior over multiple call stacks to detect frequent memory accesses. Further it was found that there were some references that occurred in every instance of the call stack, and some which occurred in just a few. Ideally, we can benefit from full context sensitivity i.e identify the exact instance of function entry or exit and accordingly evict. However, this approach has its drawbacks

1. Although we can perform any amount of offline analysis to get guidance for each specific instance it would need a large amount of information to be loaded and read during the optimized run, which is expensive.

2. Since the actual program differs from the profiling run, it is difficult to keep track of exact instances i.e. these instances might not actually occur during the final run.

Thus a the middle route was chosen where a program is cognizant of its current call stack, and looks for guidance for cache management from the results of the profiling and memory trace analysis. This is explained in more detail in Section 4.5.2. Further, there are several issues to be dealt with. The key questions to be answered are:

- When does eviction make sense and how?
- How do we detect that an object will never be referenced again or that this address will not be referenced in the near future?
- Which object to evict, and how to mark it for eviction?
- How do we indicate to the cache manager to avoid evicting a certain reference, or to readily evict it?

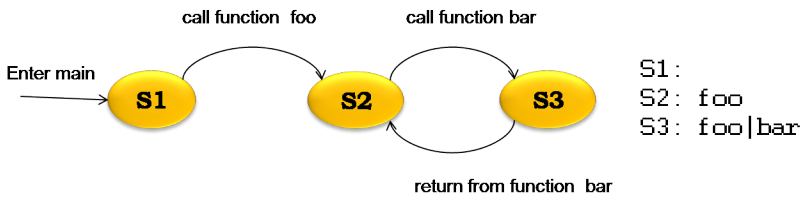

Figure 10: Program as a state machine

Thus the pin and unpin directives were introduced and these have been detailed in Section 4.5.3

4.5.2 Program as a state machine

The program itself is viewed as a Finite State Machine(FSM) in execution. This idea is depicted in Figure ẽfpic:statemachine. The State in this approach is a unique call stack. A function entry, or exit marks a change in state. However, it must be noted that a flag being set or reset within a program, or a combination of these two could also be a state. We could also allow the programmer to manually specify state change points.

The idea is to leverage this to use state transitions as decision points for the software cache manager, thus shifting the overhead to function entries and exits only, and avoiding expensive book-keeping for every dereference. Not only does this approach promise to reduce overhead, it also makes use of memory reference behavior within a certain calling context.

4.5.3 Pin and Unpin Primitives

To guide the cache manager as to which references to keep in the cache, and which ones to evict, the pin primitive has been extended in this context. A memory reference that is heavily referenced in a calling context is marked as "pinned" after it is brought into the cache. Such a reference will indicate to the cache eviction manager that it must be retained in the cache as far as possible since it is likely to be referenced in

the future. Similarly, when the memory reference is unlikely to be referenced in a calling context it can have its pin flag removed, to indicate to the cache manager that it is preferable to evict this one; when looking to make room to bring in another line. The use of these primitives in conjunction with the state machine model during run time is explained in Section 4.5.4.4.

4.5.4 Detailed Design

The program is run once in profiling mode, where the memory trace is generated, and analyzed, and a decision file is generated. The next time it reads from this file and runs an optimized run. The three main elements of the design are detailed below.

1. Instrumentation and Memory Trace Generation
2. Memory Trace Analysis
3. Optimized Run

4.5.4.1 Instrumentation and Memory Trace Generation

Trace driven analysis a popular technique for understanding memory behavior. A program can be instrumented to obtain the execution trace. They can be instrumented to log certain instructions - memory references, loads and stores, branches etc. They can be instrumented either at the source code level, assembly code level or by manipulating the application binary (source code is not required). Popular instrumentation tools for which APIs are available today are Valgrind, Pin and LLVM. However, after initial investigation, it was decided to implement a custom made instrumentation. There are no instrumentation tools that can run on the SPU of the Cell processor. The unique requirements of the program (instrumentation of the SPU code, logging remote dereferences only) necessitated the creation of custom instrumentation mechanism. Therefore, the instrumentation has been built into the Software Cache itself.

Changes in API are noted below.

Function Entry and Exit - Two new functions have been introduced that are instrumented into the code right after function entry and just before function return respectively. They mark a state transition during the final optimized run, and during the profiling run, they form the new call stack of the function and log into the trace file.

Dereferencing a remote PPU address - In a profiling run, the dereference function logs the remote PPU address to the trace file. A sample trace file has been shown in Appendix E.

4.5.4.2 Memory Trace Analysis

The memory trace analyzer assumes a raw trace file as input. It goes over the trace to aggregate behavior of memory references over unique call stacks. The detailed algorithm is as follows.

1. Create Map of references for a given call stack instance
2. Create Map of Maps for a given call stack across instances
3. Aggregate the Maps over different instances
4. Generate the decision file containing the list of references to be pinned an unpinned for each call stack based on reference probabilities.

4.5.4.3 Illustrative Example

1. Generate Memory Trace

   Consider the following trace on execution of a program on the SPU. Let the there be two functions on the SPU foo and b ar. Following is the trace generated.

The memory references in the virtual address space are actually 64 bit integers. However, they have been denoted by alphabets for the sake of simplicity in the example below.

```
ENTERED FUNCTION:foo
STACK:foo|
A
B
C
A
A
ENTERED FUNCTION:bar
STACK:foo|bar|
A
D
E
F
EXITED FUNCTION:bar
STACK:foo|
G
H
G
A
ENTERED FUNCTION:bar
STACK:foo|bar|
A
E
EXITED FUNCTION:bar
```

```
STACK:foo|
A
G
EXITED FUNCTION:foo
STACK:
```

Memory Trace Analysis

2. Create Map of references for a given call stack instance The above trace is analyzed to collapse it into a hash table, with only the unique memory references as shown below.

```
ENTERED FUNCTION: [foo]
STACK:foo|
A:3
B:1
C:1
ENTERED FUNCTION: [bar]
STACK:foo|bar|
A:1
D:1
E:1
F:1
EXITED FUNCTION: [bar]
STACK:foo|
G:2
H:1
A:1
ENTERED FUNCTION: [bar]
```

```
STACK:foo|bar|
A:1
E:1
EXITED FUNCTION: [bar]
STACK:foo|
A:1
G:1
EXITED FUNCTION: [foo]
STACK:
```

3. Create Map of Maps for a given call stack across instances These references are further collapsed into a hash table with pointers to each hash table so that they can be aggregated as shown below.

```
STACK:foo|
-----
Map0
-----
A:3
B:1
C:1
-----
Map1
-----
G:2
H:1
A:1
-----
```

```
Map2
-----
A:1
G:1

STACK:foo|bar|
-----
Map0:
-----
A:1
D:1
E:1
F:1
-----
Map1:
-----
A:1
E:1
```

4. Aggregate the Maps over different instances After this stage, for each Call Stack, the hash maps are aggregated over multiple instances to get the probability of occurence to be used to make pin and unpin decisions. This is shown below.

```
STACK:foo|
A:100.0%(3,1,1)
B:33.3% (1,0,0)
C:33.3% (1,0,0)
G:66.6% (0,2,1)
```

```
H:33.3% (0,0,1)


STACK:foo|bar|
A:100.0% (1,1)
D:50.0%  (0,1)
E:100.0% (0,1)
F:50.0%  (1,0)
```

5. Generate the decision file Using the above generated information, the decision file is generated. A threshold can be used to specify the probability above which it must be a pinned reference. In this case, consider the threshold probability to be 50

```
foo| :A,G
foo|bar| : A,E
```

The steps elaborated above to process the trace has been shown in Appendix E and Appendix E.5.

4.5.4.4 Optimized Run

Changes in API for the optimized run include the addition of new functions as listed below.

Initializing the Software Cache - A new function to load profiling information from the file is introduced which is called from the init function during optimized run. In addition to the initialization of the Software Cache itself, the decision file is read to load the pin and unpin references for each call stack into memory, through the use of a C++ STL Map. The Map uses the callstack string as the key and a structure with the corresponding in and unpin decisions as its value. for this call stack. This is loaded into memory so that look up is not expensive during program run. Let us

call this map the decisionMap.

Function Entry and Exit - The function entry and exit calls behave differently during the optimized run. During the optimized run, these functions first create the new call stack of the program by pushing or popping from the stack respectively. It then looks up the decisionMap to check if this stack exists i.e It occurred in the profile run. If yes, it loads the unpin decisions and goes through the entries in the cache to remove any pin flags on these references.

Dereferencing a remote PPU pointer - During the optimized run, when there is a true miss, the remote dereference routine, in addition to bringing in the required line, also checks to see if this reference is meant to be pinned in this context as per the decisionMap. If so, it sets the pin flag.

Making a line free - This is the main routine that implements the eviction. There is an index which keeps track of the last evicted index. This index is first incremented, and if the reference that this index points to is a pinned reference, it is skipped and index incremented. Basically, a lightweight round -robin policy is implemented, while avoiding the eviction of a pinned reference as far as possible. The pseudo code is noted below.

```
int make_free_line(void) {
    index = ++last_evicted_index;
    while( reference pointed to by index is a pinned reference)
          ++index;
    if(index == swcache_num_lines)
        index = last_evicted_index = 0;
    evict_line(index);
    return index;
}
```

This policy is beneficial in many ways.

- It uses the knowledge of memory traces to implement an eviction policy that is closer to optimal than LRU.
- Round robin is very simple and has little overhead on every remote memory dereference. All that is required needs is a last_evicted_index.
- The space requirement of LRU is circumvented as well.

4.5.5 Analysis and Performance Results

This new approach was benchmarked against the existing LRU implementation on several hand-written kernels. The nature of the kernels is that they are pointer based programs which have small dynamic memory allocation but are compute intensive. At the same time they have high memory to compute ratio, as linked list references are unpredictable and can only be brought into the cache when they are referenced, unlike arrays that could be block prefetched. Hence it is important that the DMA overhead is offset by the computation speedup. Several larger benchmarks were explored, but were deemed infeasible at this point due to reasons listed below.

- Use of C++ STL (Map) for implementation leads to code size bloat during link time. The size of the Software Cache executable with the LRU eviction implementation using STL list is 60,700 bytes upon making appropriate additions for the implementation of the run time state machine, which involved an additional functions and data structures implemented using C++ STL map, the object file increased to 292,760 bytes.For the linked list addition example used, the final executable size with the LRU version of the Software Cache is 361786 bytes and for the new policy it is 822541 bytes.
- In cases where the code compiled, dynamic memory allocation failed either during initialization of the Software Cache where space was being allocated for

Table 3: Performance Results: Linked List Addition

Number of Nodes	LRU Eviction Policy	Trace Driven Eviction Policy
500	0.004866	0.003106
1000	0.009707	0.005133
1500	0.014538	0.007362
2000	0.019347	0.010099

the Software Cache itself, or during the allocation of dynamic data structures as part of the optimization in the Software Cache or application code.

- While the implementation of a custom map data structure independent of STL was investigated, it was abandoned in the current implementation of the cache due to time constraints.

- Another restriction with the current implementation of the Software Cache is that it does not handle objects that cross line boundaries. This posed a problem for larger benchmarks.

The benchmarks used are described below and their performance results are tabulated. All benchmarks use a cache with 32 lines and a line size of 512 bytes.

Linked List Addition: This application creates four singly linked lists X,A,B and C on the PPU and passes the head pointer of all the four to the SPU. The root of the SPU function is adding(X,A,B,C) which in turn calls functions add(X,A), add(X,B), add(X,C) in sequence. The function addX adds the two linked lists passed to it. The performance results are shown in Table 3. The speedup is evident in Figure 11.

Linked List Addition and Subtraction: This application creates 3 linked lists X, A and B on the PPU and passes the head pointer of all the three to the SPU. The root of the SPU function is AddandSub(X,A,B). This function computes

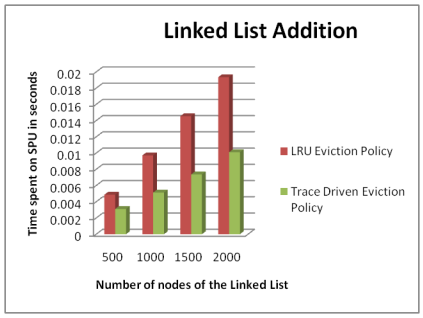

Figure 11: Linked List Addition Performance Results

Table 4: Performance Results: Linked List Addition and Subtraction

Number of Nodes	LRU Eviction Policy	Trace Driven Eviction Policy
500	0.001710	0.001561
1000	0.003260	0.002305
1500	0.004905	0.003066
2000	0.006467	0.003861

X-A and X-B.The performance results are shown in Table 4. The speedup is evident in Figure 12.

Prefix Sum Computation: This application has a central node which has next and previous pointers; each pointing to a singly linked list. A top level function decides whether to traverse to the left or the right, and then the prefix sum for each node in the singly list is computed. The performance results are shown in

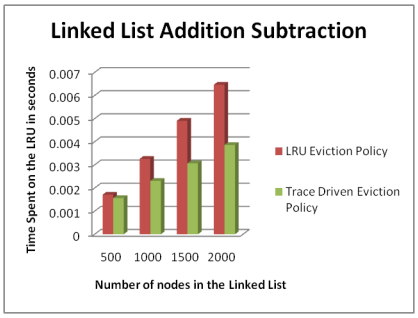

Figure 12: Linked List Addition Subtraction Performance Results

Table 5: Performance Results: Prefix Sum Computation

Number of Nodes	LRU Eviction Policy	Trace Driven Eviction Policy
500	0.180116	0.086640
1000	0.800603	0.393725

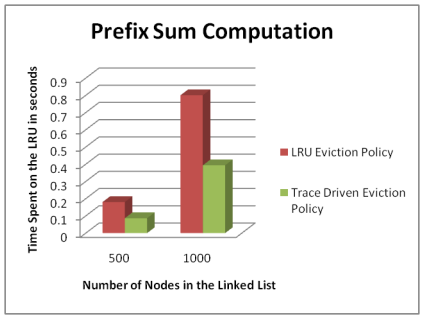

Figure 13: Prefix Sum Computation Performance Results

Table 5. The speedup is evident in Figure 13.

For number of nodes greater than 1200, a bus error was reported.

Sublist Search: This application takes 2 linked lists to check if one list is the substring of the other. The root of the SPU function is sublistSearch(X,Y) which checks if Y is a sublist of X. The performance results are shown in Table 6. The speedup is evident in Figure 14.

Tree Traversal: This application creates a tree with several nodes, and performs

Table 6: Performance Results: Sublist Search

Number of Nodes	LRU Eviction Policy	Trace Driven Eviction Policy
500	0.001038	0.001253
1000	0.002045	0.001690
1500	0.002757	0.002026
2000	0.003057	0.002238

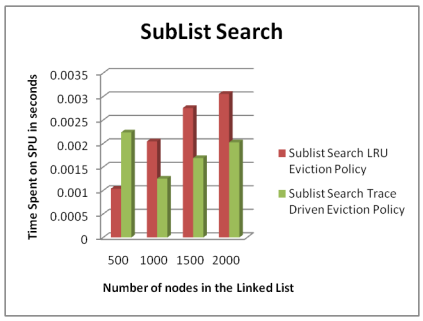

Figure 14: Sublist Search Performance Results

Table 7: Performance Results: Tree Traversal

Number of Nodes	LRU Eviction Policy	Trace Driven Eviction Policy
500	0.002041	0.001006
1000	0.002445	0.001406
1500	0.003501	0.001581
2000	0.004557	0.002026

inorder traversal of the tree on the SPU. Recursive traversal was not possible because of limited stack space on the SPU. Inorder iterative traversal was implemented. The performance results are shown in Table 7. The speedup is evident in Figure 15.

As shown in the graphs, the trace driven approach has a much better performance, and scales well to large problem sizes. This is due to the reduction in hit overhead as

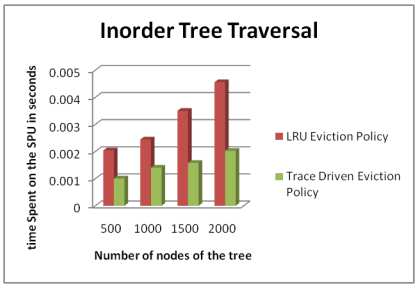

Figure 15: Tree Traversal Performance Results

well as smarter eviction. This is shown with an example in Section 4.5.6.

4.5.6 Performance Evaluation

Consider the case of linked list addition. There are 3 linked lists X,A and B created on the PPU.

```
List A: A1-->A2-->A3
List X: X1-->X2-->X3
List Y: Y1-->Y2-->Y3
```

The head of these 3 linked lists are passed to the root function on the SPU. Lets say this function is called adding. The SPU side code is:

```
adding(struct node *X, struct node *Y, struct node *A)
{
add(X,A);
add(X,B);
}
```

Consider a software controlled cache with 4 lines on the SPU. Assume that a line cannot hold more than one object for the sake of this illustration. Table 8 and Table 9 illustrate the contents of the cache at various points of execution for an LRU eviction policy. The arrow points to the reference that is Least Recently Used.

X1 ←	X1 ←	X3
A1	A1	A3
	X2	X2 ←
	A2	A2

Table 8: Cache State Changing during add(X,A) for LRU Eviction

As shown above, during execution of the routine add(X,A) the references X1,A1, X2, A2, X3 and A3 are accessed in that order by evicting the LRU reference whenever it did not exist in the cache. During execution of add(X,B) X1,B1,X2,B2,X3,B3 are

X3 ←	X2	X2 ←
A3	B2	B2
X1	X1 ←	X3
B1	B1	B3

Table 9: Cache State Changing during add(X,B) for LRU Eviction

again brought into the cache in that order. Thus it is observed that it suffered 8 misses.

Table 10 and 11 illustrate the changing cache contents for the trace driven eviction policy. It is assumed that profiling has indicated that the references in linked list X have been referenced heavily and have been marked for pinning. The pinned references in the cache are indicated by a • beside it.

X1 •	X1 •	X1 •
A1	A1	X3 •
	X2 •	X2 •
	A2	A3

Table 10: Cache State Changing during add(X,A) for Trace Driven Eviction

X1 •	X1 •	X1 •
X3 •	X3 •	X3 •
X2 •	X2 •	X2 •
B1	B2	B3

Table 11: Cache State Changing during add(X,B) for Trace Driven Eviction

As depicted above, in the trace driven policy references X1, A1, X2, A2, X3, A3 are first accessed during the execution of add(X,A). On bringing in references X1, X2 and X3 into the cache, they remain in the cache as "pinned" references as suggested by the decision file generated at the end of the profile run. When add(X,B) is executed, it looks first for X1 and B1. X1 is already in the cache, and A3 is evicted to bring in B1. Similarly, B2 and B3 are brought into the last cache line. Thus we can see that it suffered 4 misses.

Table 12: LRU Overheads Breakup

Overhead	**Time in seconds**
Compulsory Miss	0.000066
Capacity Miss	0.000052
Hit	0.000036

Table 13: Trace Driven Policy Overheads Breakup

Overhead	**Time in seconds**
Compulsory Miss	0.000056
Capacity Miss	0.000040
Hit	0.000020

The breakup of overheads for LRU have been detailed in Table 12 and those for the Trace Driven Policy are in Table 13. These have been qualitatively explained previously.

4.5.7 Design Tradeoffs

4.5.7.1 Solution Space

The solution space for this problem consists of various degrees of tracking. Ideally, we can perform more rigorous analysis of the memory traces to have more sensitive run time behavior. For instance:

- We can exploit the call stack further to see how memory behavior changes with changing call stack. If we have a function, say f that calls g and p or q conditionally , we could analyze the trace to make decisions at every point, as to whether a reference X that was pinned at g, is always referenced if it calls p, and never if it calls q, and so on. In cases where an address is referenced on and off, the minimal stack configuration where a decision can be made could be computed. This would let us create a probability distribution to continually build the list of references that need to be pinned. However, this requires

the maintenance of additional data structures, and is memory intensive. The computation involved is also an overhead that needs to be offset by a suitably predictable long running benchmark.

- Instead of aggregating behavior of call stacks, we can treat each unique function call as a separate instance. This would provide for more context sensitive information. However, the number of function calls are normally huge as depicted in table. While we can afford to perform any amount of offline analysis, such tracking would lead to an overhead during run time and might infact be unnecessary.

4.5.7.2 Balancing Time and Space Constraints

More rigorous run time analysis might end up being an overhead rather than a benefit. The limitations of the Cell SPU and its tight memory constraints leaves little space for storing large dynamic data structures needed for the purpose of analysis. Therefore, we need to strike a balance and use a light weight eviction policy while still retaining the benefits of memory profiling. When compilation was attempted for a large program, the compiler returned with errors as its object code size exceeded the local memory of the SPU. This has been shown in Appendex F. Similarly, the use of this new eviction policy should not increase the number of branches on the SPU. The SPU performs poorly with code that has a large number of branches because it lacks a Branch Prediction Unit. In this case, the problem has been overcome by the usage of branch hints, indicating that a reference is "unlikely" to be pinned, when the eviction manager checks to see if a reference is pinned before it decides to skip over to the next index in the cache. Another point is that while providing maximum flexibility and reconfigurability of the Software Cache, allowing the number or percentage of references to be pinned must be controlled. Too many pinned references in the cache might end up degrading performance as the cache eviction manager would

continually loop over the cache entries to find an unpinned line. In the examples cited, the percentage of pinned references is limited to overcome this problem.

4.6 Conclusion

4.6.1 Contribution

The main contributions of this work are listed below.

- System to seamlessly converting C code into SPU and PPU code for the Cell. SPU and PPU side code are generated for the parittions suggested by the toolchain.
- Novel memory trace analysis with the view of a program as Finite State Machine, where an offline engine generates guidance for run-time state machine transitions.
- Extension and use of the concept of pin and unpin in this context
- Implementation of a cache eviction policy which interweaves a lightweight round robin policy with memory trace findings.
- Exploration of the solution space of various possibilities to solve the same problem using memory profiling, through partial or full implementations of the same. These provided useful insights into the constraints of the Cell and the SPU in particular.
- This implementation significantly reduces overhead in pointer intensive code due to its irregularity. Array references on the other hand are predictable, and prefetching can be used to get a speedup.
- Cell does not have many libraries ported on to it. This approach, combined with GLIMPSES for analyzing program behavior, can be used to port several existing libraries to be used on the Cell.

4.6.2 Future Work

Things that need to be explored in the future are:

- Implementation of custom data structures to avoid C++ Standard Template Library. This could reduce object code size and pave way for storing other data structures at run time, or allocating more space for the cache itself.

- The current implementation uses a Software Cache which cannot handle an object spanning a line boundary. This limited the set of benchmarks that this trace driven policy was tested on. Use of a Cache that could handle such objects could increase the scope of the project.

APPENDIX A

LIST OF ABBREVIATIONS

Cell	Cell Broadband Engine
STI	Sony Toshiba IBM
PPU	Power Processor Unit
PPE	Power Processor Engine
SPU	Synergistic Processor Unit
SPE	Synergistic Processor Engine
SIMD	Single Instruction Stream Multiple Data Stream
STL	Standard Template Library
LRU	Least Recently Used
MRU	Most Recently Used
RISC	Reduced Instruction Set Instruction
API	Application Programmer Interface
DMA	Direct Memory Access
OS	Operating System
FPR	Floating Point Unit
EIB	Element Interconnect Bus
CFG	Control Flow Graph
LLVM	Low Level Virtual Machine
CISC	Complex Instruction Set Architecture
RISC	Reduced Instruction Set Architecture
SDK	Software Development Kit
PS3	Play Station 3
API	Application Programmer Interface
LS	Local Store
STL	Standard Template Library

APPENDIX B

GLIMPSES USE CASES

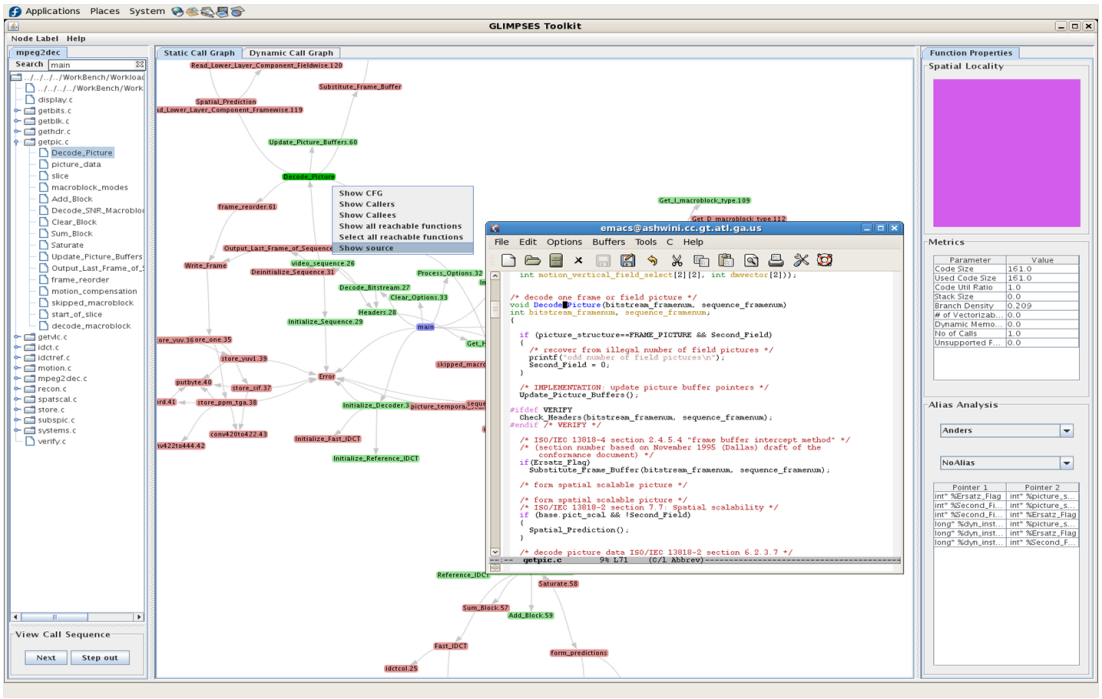

Figure 16: GLIMPSES: Editing Source Code

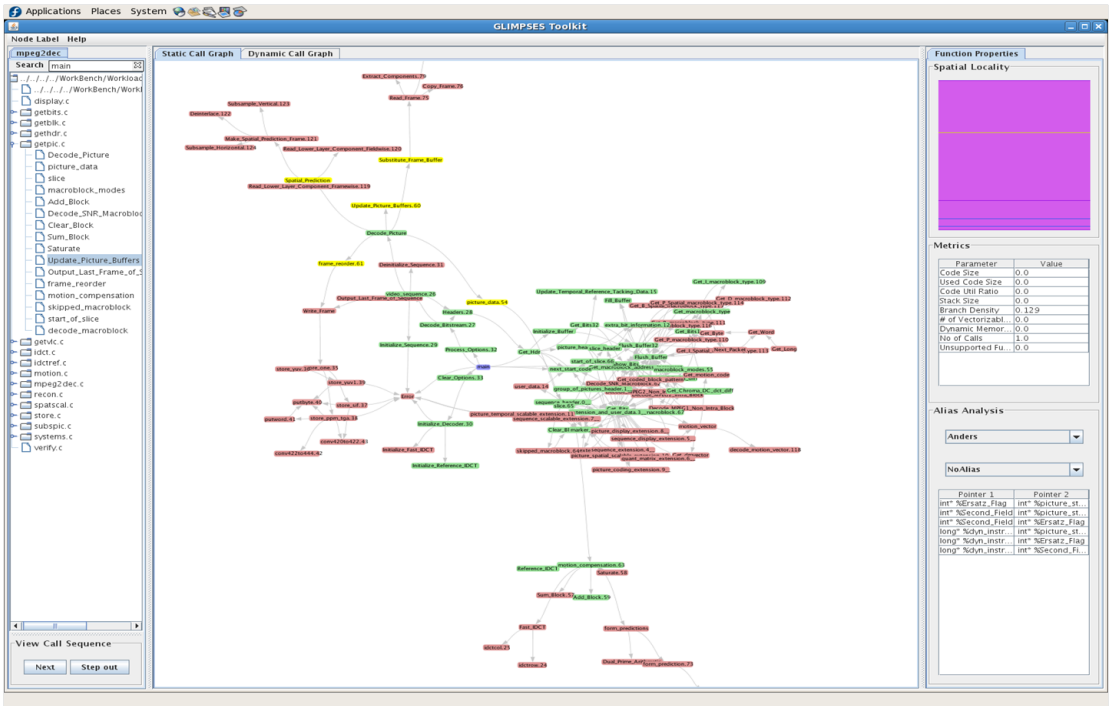

Figure 17: GLIMPSES: Viewing Caller Callee Relationships

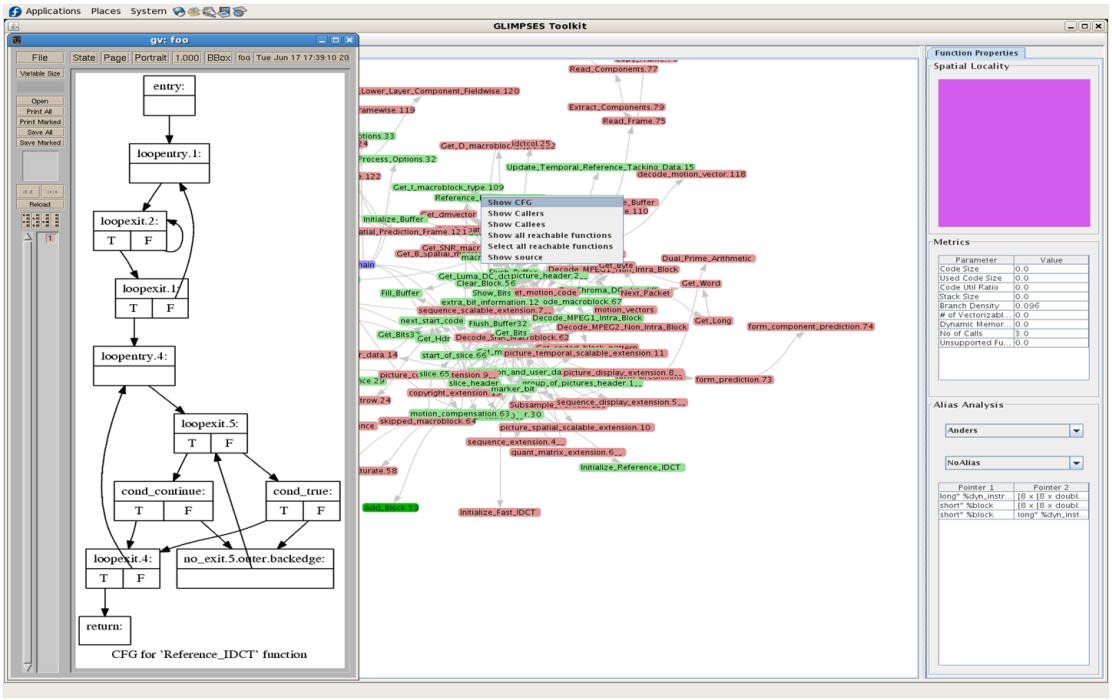

Figure 18: GLIMPSES: Viewing the Control Flow Graph of a function

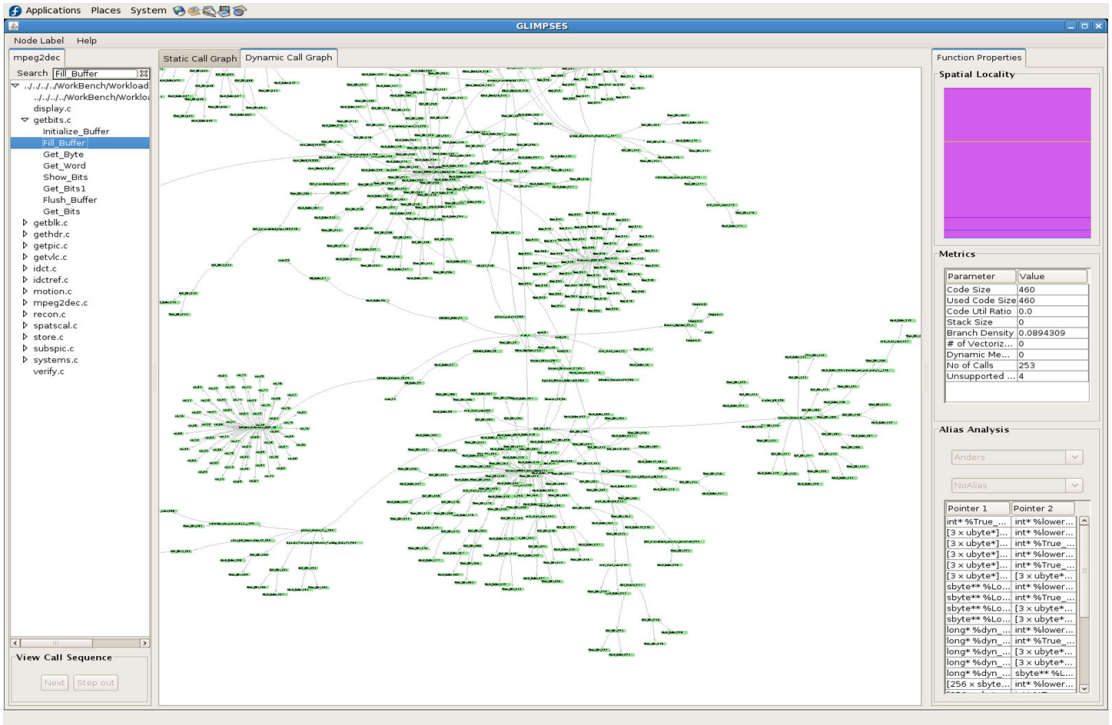

Figure 19: GLIMPSES: Viewing the Dynamic Call Graph of the program

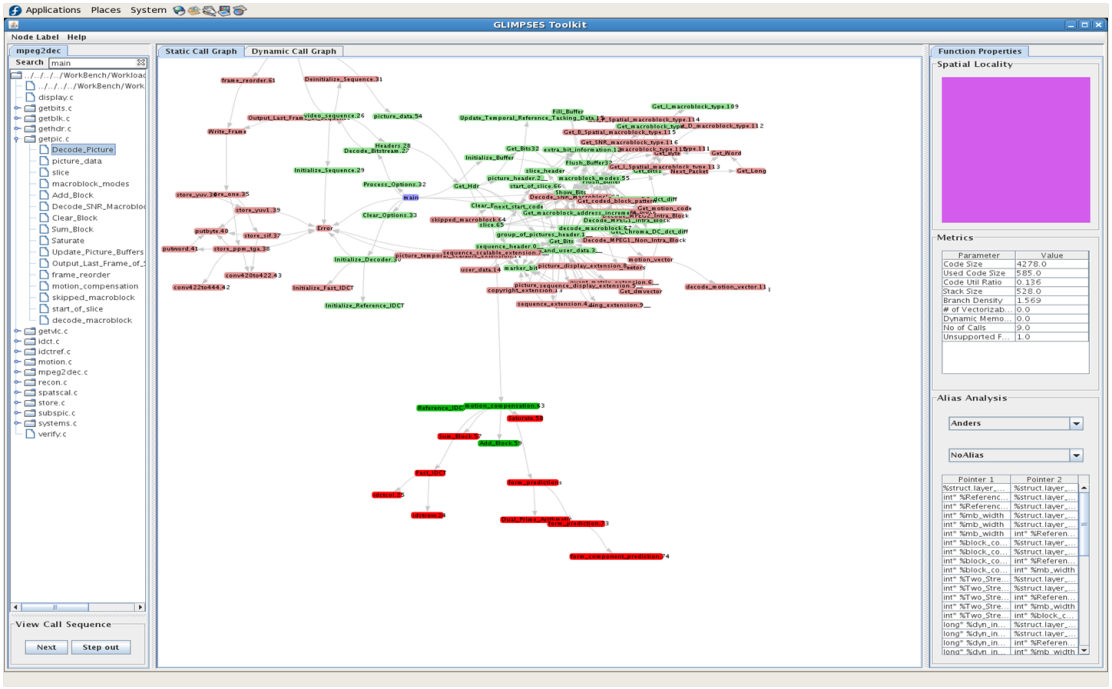

Figure 20: GLIMPSES: Viewing all reachable functions

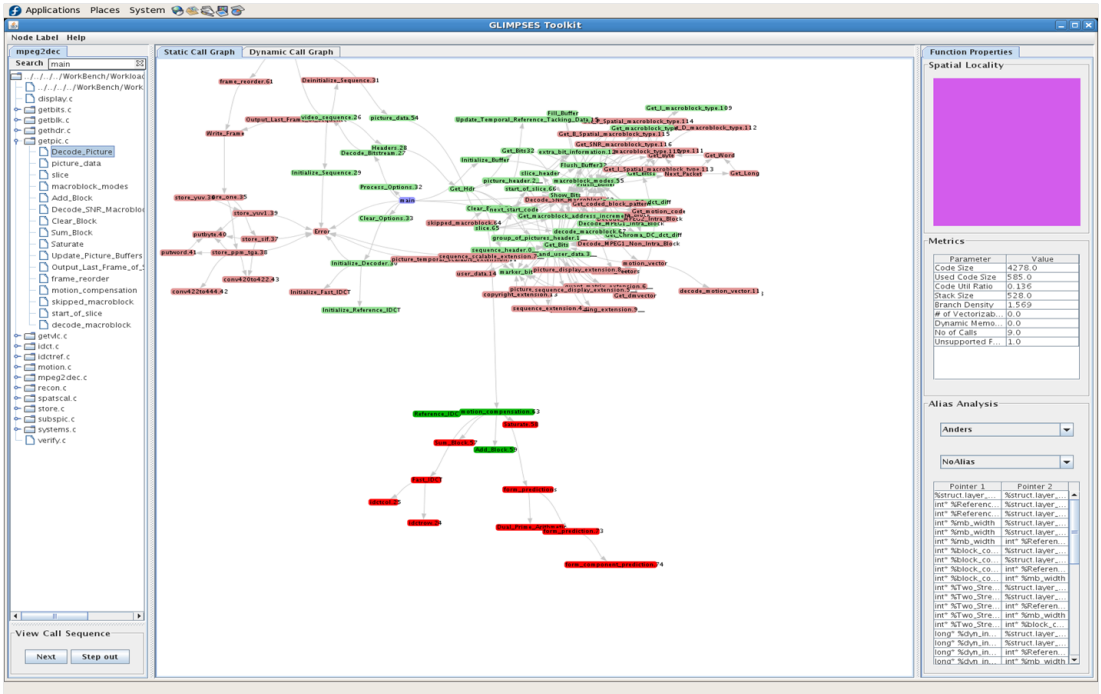

Figure 21: GLIMPSES: Viewing Static and Dynamic Graph

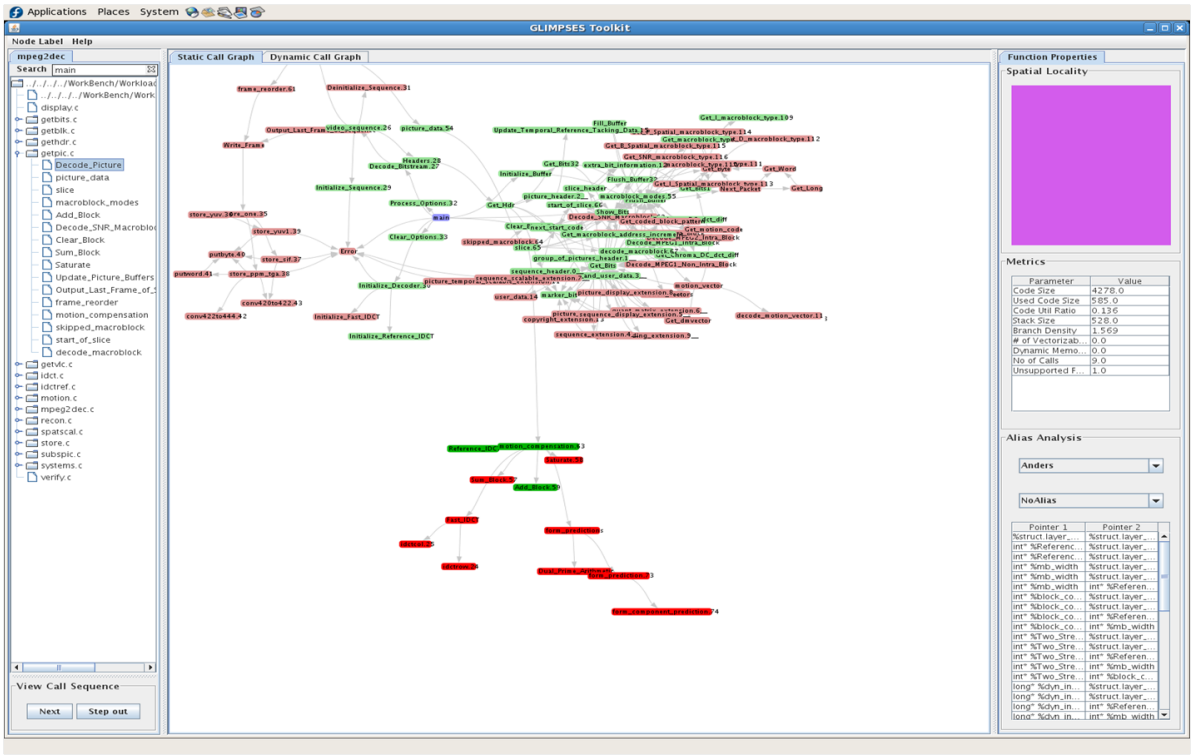

Figure 22: GLIMPSES: Searching for a function

APPENDIX C

GLIMPSES SAMPLE XML FILE

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
<graph edgedefault="directed">
<!-- data schema -->
<key id="name" for="node" attr.name="name" attr.type="string"/>
<key id="codesize" for="node" attr.name="codesize" attr.type="string"/>
<key id="stacksize" for="node" attr.name="stacksize" attr.type="string"/>
<key id="brfraction" for="node" attr.name="brfraction" attr.type="string"/>
<key id="avloops" for="node" attr.name="avloops" attr.type="string"/>
<key id="mallocsize" for="node" attr.name="mallocsize" attr.type="string"/>
<key id="external" for="node" attr.name="external" attr.type="int"/>
<key id="hasunsupported" for="node" attr.name="hasunsupported" attr.type="string"/>
<key id="lslimithit" for="node" attr.name="lslimithit" attr.type="string"/>
<key id="calls" for="node" attr.name="calls" attr.type="int"/>
<key id="rank" for="node" attr.name="rank" attr.type="string"/>
<!-- testfgmain-->
<node id="6">
<data key="name">f</data>
<data key="codesize">86</data>
<data key="stacksize">896</data>
<data key="brfraction">0</data>
<data key="avloops">0</data>
<data key="mallocsize"></data>
<data key="external">0</data>
<data key="hasunsupported">0</data>
<data key="lslimithit"></data>
<data key="calls">2</data>
<data key="rank">982</data>
</node>
<node id="2">
<data key="name">g</data>
<data key="codesize">335</data>
<data key="stacksize">1024</data>
<data key="brfraction">0.047619</data>
```

```
<data key="avloops">0</data>
<data key="mallocsize"></data>
<data key="external">0</data>
<data key="hasunsupported">4</data>
<data key="lslimithit"></data>
<data key="calls">2</data>
<data key="rank">2341.0238095</data>
</node>
<node id="0">
<data key="name">main</data>
<data key="codesize">152</data>
<data key="stacksize">512</data>
<data key="brfraction">0.0666667</data>
<data key="avloops">0</data>
<data key="mallocsize"></data>
<data key="external">0</data>
<data key="hasunsupported">2</data>
<data key="lslimithit"></data>
</node>
<node id="4">
<data key="name">test</data>
<data key="codesize">129</data>
<data key="stacksize">128</data>
<data key="brfraction">0</data>
<data key="avloops">0</data>
<data key="mallocsize"></data>
<data key="external">0</data>
<data key="hasunsupported">1</data>
<data key="lslimithit"></data>
<data key="calls">1</data>
<data key="rank">257</data>
</node>
<edge source="0" target="2"></edge>
<edge source="0" target="4"></edge>
<edge source="2" target="6"></edge>
</graph>
</graphml>
```

APPENDIX D

PORTING CODE TO CELL

D.1 Original Code

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int ret;
    a= 5;
    b= 10;
    ret = find_sum(a,b);
    printf("Sum is %d\n", ret);
    return 0;
}

int findsum(int a, int b)
{
        int sum = a+b;
        return sum;
}
```

D.2 Converted Code

D.2.1 Control Block

```
#ifndef SIMPLE_COMMON_H
#define SIMPLE_COMMON_H

typedef struct context_block {
    int a;
    int b;
} context_block __attribute__ ((aligned (128)));

#endif
```

D.2.2 PPU Code

PPU CODE: simple.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <libspe2.h>
#include "simple_common.h"
#include "prt.ppu.h"

struct context_block ctx __attribute__ ((aligned (128)));
extern spe_program_handle_t findsum_spe_prog;

int main()
{
    int ret;
    ctx.a= 5;
    ctx.b= 10;
    ret = prt_run((uint64_t) &ctx, & findsum_spe_prog);
    printf("Sum is %d\n", ret); return 0;
}
```

D.2.3 SPU Code

SPU CODE: findsum.cpp

```
#include "prt.spu.h"
#include "simple_common.h"
#include "remote_ptr.hpp"

int findsum(int a, int b);

int prt_main(uint64_t ea)
{
    //Initialize the software cache
    swcache_init(32, 512);

    //Wrap the control block pointer into a remote_ptr
    remote_ptr<struct context_block> cb(ea);

    int idx = findsum(cb->a, cb->b);

    //Software cache cleanup
    swcache_cleanup();
    return idx;
```

```
}

int findsum(int a, int b)
{
        int sum = a+b;
        return sum;
}
```

D.2.4 Makefile

```
CC=ppu-g++
SPU_CC=spu-g++
PPU_EMBEDSPU=ppu-embedspu
PPU_AR=ppu-ar
LIBPRT=../../../libprt
SWCACHE=../../../software_cache
CFLAGS=-I$(LIBPRT)
LDFLAGS=$(LIBPRT)/prt.ppu.o -lspe2 -lpthread

all: simple.cpp partitions.a
        $(CC) $(CFLAGS) $(LDFLAGS) simple.cpp partitions.a -o simple

clean:
        rm -f *.o *.a simple

partitions.a: findsum-embed.o
        $(PPU_AR) -qcs partitions.a findsum-embed.o

findsum-embed.o: findsum.cpp
        $(SPU_CC) $(CFLAGS) -I$(SWCACHE) $(LIBPRT)/prt.spu.o $(SWCACHE)/swcache.o findsum.cpp -o findsum
        $(PPU_EMBEDSPU) findsum_spe_prog findsum findsum-embed.o
```

APPENDIX E

MEMORY TRACE ANALYSIS

E.1 Raw trace

```
0x100d5980
ENTERED FUNCTION: [findNode|]
STACK:findNode|
0x100d5700
0x100d5780
0x100d5800
0x100d5880
0x100d5900
EXITED FUNCTION: [findNode|]
STACK:
```

E.2 Extracted Hash Maps of Probabilities

```
ENTERED FUNCTION: [findNode|]
STACK:findNode|
0x100d5780:1
0x100d5700:1
0x100d5800:1
0x100d5900:1
0x100d5880:1
EXITED FUNCTION: [findNode|]
STACK:
```

E.3 Combining Maps

```
===STACK:findNode|===[1 instances ]


Map 0
0x100d5700:1
0x100d5780:1
0x100d5800:1
0x100d5880:1
```

```
0x100d5900:1
```

E.4 Aggregated Probabilities

```
[Summary]

Total call stacks : 2
Total unique call stacks   : 1


===STACK:findNode|===[1 instances ]

0x100d5700-100% (1,)
0x100d5780-100% (1,)
0x100d5800-100% (1,)
0x100d5880-100% (1,)
0x100d5900-100% (1,)
```

E.5 Decision File

```
STACK:findNode| 100d5700,100d5780,100d5800,100d5880,100d5900,
```

APPENDIX F

SPU MEMORY CONSTRAINT ERRORS

```
/usr/lib/gcc/spu/4.1.1/../../../../spu/bin/ld: .text exceeds local store range
/usr/lib/gcc/spu/4.1.1/crtbegin.o: In function `__do_global_dtors_aux':
crtstuff.c:(.text+0xc): relocation truncated to fit: SPU_REL16 against `.bss'
crtstuff.c:(.text+0x34): relocation truncated to fit: SPU_REL16 against `.data'
crtstuff.c:(.text+0x3c): relocation truncated to fit: SPU_REL16 against `.data'
crtstuff.c:(.text+0x54): relocation truncated to fit: SPU_REL16 against `.bss'
/usr/lib/gcc/spu/4.1.1/crtbegin.o: In function `frame_dummy':
crtstuff.c:(.text+0x80): relocation truncated to fit: SPU_ADDR18 against `.jcr'
crtstuff.c:(.text+0x84): relocation truncated to fit: SPU_REL16 against `.jcr'
/usr/lib/gcc/spu/4.1.1/crtbegin.o:(.fini+0x0): relocation truncated to fit: SPU_REL16 against `__do_global
/usr/lib/gcc/spu/4.1.1/crtend.o: In function `__do_global_ctors_aux':
crtstuff.c:(.text+0x8): relocation truncated to fit: SPU_ADDR18 against `.ctors'
/usr/lib/gcc/spu/4.1.1/crtend.o:(.init+0x0): relocation truncated to fit: SPU_REL16 against `__do_global_c
/usr/lib/gcc/spu/4.1.1/../../../../spu/lib/crt1.o: In function `_start':
(.text+0x0): relocation truncated to fit: SPU_REL16 against symbol `__ea_local_store' defined in COMMON se
/usr/lib/gcc/spu/4.1.1/../../../../spu/lib/crt1.o: In function `_start':
(.text+0x28): additional relocation overflows omitted from the output
/usr/lib/gcc/spu/4.1.1/libstdc++.a(basic_file.o): In function `std::__basic_file<char>::sys_open(int, std
/root/sdk3.0rebuild/spu-gcc/BUILD/spu-gcc-4.1.1/obj-spu/spu/libstdc++-v3/src/basic_file.cc:212: undefined
collect2: ld returned 1 exit status
```

REFERENCES

[1] "Cache algorithms." http://en.wikipedia.org/wiki/Cache_algorithms[Date Accessed: May 10, 2009].

[2] "Cell microprocessor." http://en.wikipedia.org/wiki/Cell_(microprocessor) [Date Accessed: May 10, 2009].

[3] "developerworks : Cell broadband engine resource center." http://www.ibm.com/developerworks/power/cell/[Date Accessed: May 10, 2009].

[4] "Fixstars – optimized solutions for multi-core processors cell, gpu; linux for cell, playstation, ps3, qs22, and mercury." http://us.fixstars.com/[Date Accessed: May 10, 2009].

[5] "The graphml file format." http://graphml.graphdrawing.org/[Date Accessed: May 10, 2009].

[6] "Intel vtune." http://software.intel.com/en-us/intel-vtune/[Date Accessed: May 10, 2009].

[7] "An introduction to gcc using the profiler gprof." http://www.network-theory.co.uk/docs/gccintro/gccintro_80.html[Date Accessed: May 10, 2009].

[8] "The llvm compiler infrastructure." http://www.llvm.org[Date Accessed: May 10, 2009].

[9] "Page replacement algorithms." http://en.wikipedia.org/wiki/Page_replacement_algorithm[Date Accessed: May 10, 2009].

[10] "Prefuse interactive information visualization toolkit." http://prefuse.org/[Date Accessed: May 10, 2009].

[11] "Prefuse interactive information visualization toolkit." http://prefuse.org/[Date Accessed: May 10, 2009].

[12] "Ptp eclipse parallel tools platform." http://www.eclipse.org/ptp/[Date Accessed: May 10, 2009].

[13] "Valgrind home." http://valgrind.org/[Date Accessed: May 10, 2009].

[14] "Vtune." http://en.wikipedia.org/wiki/Vtune[Date Accessed: May 10, 2009].

[15] Blachford, N., "Cell architecture explained version 2," 2005. http://www.blachford.info/computer/Cell/Cell0_v2.html[Date Accessed: May 10, 2009].

[16] Ganapathy Senthil, Sasikanth Gudla, P. K. B., "Exploring software cache on the cell be processor," 1986.

[17] Hoftsee, H. P., "Power efficient processor architecture and the cell processor,"

[18] Jaswanth Sreeram, S. P., "Glimpses: A profiling tool for rapid spe code prototyping,"

[19] Shauer, B., "Multicore processors a necessity." http://www.csa.com/discoveryguides/multicore/review.php?SID=tisu4dt0hcao8vjfc9t47nl192[Date Accessed: May 10, 2009].