SCRATCH-PAD MEMORY BASED CUSTOM PROCESSOR DESIGN FOR GRAPH APPLICATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Gülce Pulat
September 2020

SCRATCH-PAD MEMORY BASED CUSTOM PROCESSOR DESIGN FOR GRAPH APPLICATIONS
By Gülce Pulat
September 2020

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Özcan Öztürk(Advisor)

Süleyman Tosun

Uğur Güdükbay

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

SCRATCH-PAD MEMORY BASED CUSTOM PROCESSOR DESIGN FOR GRAPH APPLICATIONS

Gülce Pulat
M.S. in Computer Engineering
Advisor: Özcan Öztürk
September 2020

As more and more domains have started to process ever-growing graphs, the importance of graph analytics applications became more apparent. However, general-purpose processors are challenged to deal with the large memory footprint and the associated random memory accesses in graph applications, directing researchers towards domain-specific solutions. In this dissertation, we present a custom RISC-V graph processor that tries to increase the performance of graph applications by reducing the memory accesses. The novelty of the graph processor lies in the design of our software-controlled scratch-pad memories: Edge Scratch-Pad (ESP), Vertex Scratch-Pad (VSP), and Global Scratch-Pad (GSP). While ESP is preloaded with the edge data in parallel with the execution, VSP relieves the vertex traffic by reducing the conflicts caused by the vertex-related memory accesses. GSP takes over the load of the rest of the memory accesses as these three SPMs replace the conventional caches found in general-purpose systems. For the software to control this new functionality embedded in the graph processor, we extended RISC-V instruction set architecture with custom SPM-related instructions. We provided compiler support for the instructions and we modified the widely used PageRank, Single-Source Shortest Path, and Breadth-First Search algorithms in graph processor fashion to demonstrate the software-hardware interaction needed for the design. The experimental results on these applications show that the graph processor makes 18% to 72% less datapath-blocking memory accesses compared to a general-purpose processor based on the same RISC-V core.

Keywords: Iterative Graph Applications, Domain-Specific Architectures, Custom Processor, Instruction Set Architecture, RISC-V.

ÖZET

ÇİZGE UYGULAMALARI İÇİN MÜSVEDDE BELLEK TEMELLİ ÖZEL İŞLEMCİ TASARIMI

Gülce Pulat
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özcan Öztürk
Eylül 2020

Çizge analizi uygulamalarının önemi, giderek daha çok alanın sürekli büyüyen çizge yapılarını işlemeye başlamasıyla daha da belirgin hale gelmiştir. Bununla birlikte, genel amaçlı işlemciler çizge uygulamalarının geniş bellek ayak izi ve rastgele bellek erişimleri karşısında zorlanmaktadır. Dolayısıyla araştırmacılar alana özel donanım içerikli çözümlere yönelmiştir. Bu tezde, çizge uygulamalarının performansını bellek erişimlerini azaltarak artırmaya çalışan özel bir RISC-V çizge işlemcisi sunulmaktadır. Çizge işlemcisinin yeniliği içerdiği yazılım kontrollü Ayrıt Müsvedde (ESP), Düğüm Müsvedde (VSP) ve Global Müsvedde (GSP) isimli müsvedde belleklerine (scratch-pad memories - SPMler) dayanmaktadır. SPMler genel amaçlı sistemlerde bulunan geleneksel ön-belleklerin yerini almak için tasarlanmıştır. ESP çekirdeğin işlemlerine paralel olarak ayrıt verisiyle önceden yüklenirken, VSP düğüm ile ilgili bellek erişimlerinin neden olduğu çakışmaları azaltarak düğüm trafiğini hafifletir. GSP ise geriye kalan bellek erişimlerinin yükünü üstlenmektedir. Çizge işlemcisine gömülü bu yeni işlevselliğin yazılım tarafından kontrol edilebilir olması için RISC-V komut kümesi mimarisi SPM ile ilgili özel komutlarla genişletilmiştir. Komutların ihtiyaç duyduğu derleyici desteği de sağlanmıştır. Tasarımın gerektirdiği yazılım-donanım etkileşimini göstermek için yaygın olarak kullanılan PageRank, Tek Kaynaklı En Kısa Yol ve Sığ Öncelikli Arama algoritmaları çizge işlemcisi tarzında değiştirilmiştir. Bu uygulamalarla yapılan deneylerin sonuçları, çizge işlemcisinin aynı RISC-V çekirdeğini kullanan genel amaçlı bir işlemciye kıyasla veri yolunu durdurucu bellek erişimlerini yüzde 18 ile yüzde 72 daha az yaptığını göstermektedir.

Anahtar sözcükler: Yinelemeli Çizge Uygulamaları, Alana Özel Mimariler, Özel İşlemci, Komut Kümesi Mimarisi, RISC-V.

Acknowledgement

This work has been supported in part by Scientific and Technological Research Council of Turkey (TÜBİTAK) 1001 program through the EEEAG 119E559 project.

I would like to thank my advisor Özcan Öztürk for his patience and support. I am grateful to him for giving me the chance to work on what I have been the most curious about.

I would like to thank the members of the committee, Süleyman Tosun and Uğur Güdükbay, for sparing the time to evaluate this work.

As I complete my studies in Bilkent, I want to recognize some of my professors: Mustafa Nakeeb, Ceyhun Bulutay, Andrew Ploeg, Simon Wigley, and Orhan Aytür. I will do my best to always remember the lessons they gave.

Finally, I would like to express my gratitude to my dear family and friends. How I happen to have so many nice people in my life is a little bit beyond me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are widely used in many domains such as robotics, social network analysis, computational biology, and machine learning [1, 2]. Example algorithms include Single-Source Shortest Path (SSSP) for cognitive systems, PageRank (PR) [3] used in search engines for ordering hyperlinks, Betweenness Centrality (BC) [4] helping to analyze social networks, Alternating Least Squares (ALS) [5] as a method of matrix factoring in personal recommendation systems, and Breadth-First Search (BFS) which deserves our attention as a building block in many graph applications [2]. In the era of big data, not only the number of domains using graph algorithms but also the size of the data abstracted by graphs is continuously increasing. This abundance, as well as the size of graphs, emphasize the significance of efficient graph processing and we are thus motivated to better understand and improve the performance of graph applications.

On the other hand, due to the end of Moore's Law, we see a decline in the rate with which the computing capability used to enhance. Conventional hardware platforms have started to fail to support the special needs of applications working on the data that is even bigger than ever. This is one of the main reasons why recently there has been a shift from general-purpose to application-specific hardware. Different areas adopting such mechanisms include machine learning [6], digital signal processing [7], and graph processing [8]. With its large memory

footprint, graph processing is notoriously troublesome as graph applications suffer from the poor locality, have communication-centric workloads, and frequently need to make irregular random memory accesses [9, 10]. These features cause the memory accesses to be one of the biggest challenges in graph processing [8,11–13]. Therefore, while suggesting optimizations for the graph applications or the underlying hardware, this memory bottleneck becomes a part of the discussion and attempts to reduce the memory accesses naturally follow.

This work endeavors to develop a domain-specific processor for graph applications. Highlighting the memory bottleneck, we extend the RISC-V instruction set architecture (ISA) [14] and we present a scratch-pad memory (SPM) based graph processor where iterative graph algorithms can execute with as little datapath-blocking memory accesses as possible. We offer a minimal design in the hope that the architecture proves useful even under the most modest conditions and is eventually suitable for a broader class of processors. Finally, we keep our scope in processor-only limits, promoting ease of programmability and differentiating from the previous work done on heterogeneous systems [8, 15, 16].

1.1 Objective of the Thesis

In this dissertation, we try to provide a scratch-pad memory (SPM) based domain-specific processor that aims to improve the performance of iterative and vertex-centric graph applications by reducing the number of datapath-blocking memory accesses. Instead of the conventional caches found in general-purpose CPUs, we present software-controlled SPMs along with the necessary architectural and software-level support to deliver higher hit rates.

The main contributions of our work in constructing the graph processor are the following:

- We design custom SPMs considering the common features of the memory traffic in graph applications. Our proposed architecture features the Edge

Scratch-Pad (ESP) for the edge traffic, the Vertex Scratch-Pad (VSP) for the vertex data, and the Global Scratch-Pad (GSP) handling the rest of the memory accesses.

- We extend RISC-V instruction set architecture with custom instructions to control and communicate with SPMs.
- We extend and modify an open-source RISC-V core to transform it into a custom graph processor that is equipped with the aforementioned SPMs and control units.
- We provide compiler support for the new instructions and demonstrate how the newly developed functionality can be employed in widely used graph algorithms such as PageRank, Single-Source Shortest Path, and Breadth-First Search.
- Lastly, we evaluate the performance of the graph processor, mainly focusing on the miss rates and main memory accesses under different conditions.

1.2 Organization of the Thesis

Following the introduction in this chapter, the thesis continues with Chapter 2 where we investigate the previous work done on graph analytics by examining software frameworks and hardware optimizations separately. Next, Chapter 3 introduces the RISC-V instruction set architecture and the Ibex core, provides a brief background on scratch-pad memories, and summarizes the algorithmic characteristics of graph applications.

In Chapter 4, the suggested graph processor architecture is described in detail. We explain the architectural features and the control mechanisms of our custom units, giving motivations along the way. On the software front, Chapter 5 complements Chapter 4 with formal definitions of the new instructions. It also contains modified PageRank, Single-Source Shortest Path, and Breadth-First Search algorithms, providing insight on how the high-level code should be written with

the graph processor in mind. At the end of the chapter, we shortly illustrate our framework to complete the discussion regarding the architecture and the software.

Chapter 6 is dedicated to the experimental setup and results. We compare our design against a general-purpose processor in terms of miss rates and datapath-blocking memory accesses, and we shed further light on some architectural features of the graph processor with utilization and sensitivity analysis. We discuss our findings, mention the merit and the drawbacks of our design, and offer guidance on possible future work in Chapter 7. Finally, we conclude the thesis with a brief summary of the work done in Chapter 8.

Chapter 2

Related Work

As the need for graph analytics started to grow, a considerable amount of research effort has been put to improve the performance of graph applications. The efforts can be categorized as software frameworks and hardware optimizations targeted for accelerators, processors, or heterogeneous systems where the two are merged. This chapter is dedicated to summarizing the previous work done in this context.

2.1 Software Implementations

At the software level, previous studies provide various graph engines where applications are optimized in different ways. The options related to the design of software include but are not limited to work activation, data flow, and execution order, while the underlying memory architecture can be distributed or shared. For example, Pregel [17] is proposed for distributed systems. It assigns vertices across different machines organized in separate clusters where each vertex can run its own function in a push-based manner at each iteration. In contrast, GraphLab [18] adapts a pull-based data flow where users are able to define their own Gather-Apply-Scatter (GAS) function and have control over the execution order. Initially

appearing as a counterpart to Pregel, Giraph similarly targets distributed systems but it performs asynchronous execution and features some advancements such as multi-threading and out-of-core computation [19]. One final example is GraphCHi [20], which is a large-scale graph computation system. Specifically, it separates large graphs into smaller parts and offers novel parallelization methods on shared Flash/SSD platforms. The majority of these systems take a vertex-centric approach where vertices perform local computations on their neighbors or edges. Within this vertex-centric computation, execution continues iteratively until the convergence is reached. Aside from being vertex-centric, these engines do not feature common design choices or platform specifications, which constitutes a problem of standardization.

These frameworks had been proposed as software solutions to be executed on general-purpose processors. Hence, they are different from the approach taken in this work in the sense that they do not seek to alter the architecture of the processor in any way.

2.2 Accelerator-Level Optimizations

Heterogeneous systems where accelerators collaborate with each other or FPGA blocks and processors are popular for graph processing. In one previous work, Yesil et al. proposed accelerator blocks for vertex-centric iterative graph applications [16]. In their work, each accelerator unit utilizes separate caches for the edge, vertex, and activity informations and it includes specialized blocks to perform the widely adopted Gather-Apply-Scatter (GAS) function. In such a design, many accelerator units can be instantiated together and communicate via a network. In a related work, Zhou and Prasanna [21] reported that vertex-centric approaches result in too many random accesses whereas edge-centric schemes produce redundant reads. Therefore, they presented a hybrid paradigm combining the two for improved parallelism in heterogeneous platforms.

In another notable work, Kapre provided a 3-stage soft-processor specialized

to support a custom ISA including new instructions corresponding to send, receive, accumulate, and update operations commonly performed in sparse graph applications [11]. The implementation gathers many of such soft cores in a network referred to as "Graph SoC Accelerator". In this architecture, the execute stages of the soft cores involved are customized to better function in the Bulk Synchronous Parallel (BSP) paradigm. The specialized components are able to continuously perform typical graph operations like the accumulation of edges and spreading the information about nodes. The performance is further accelerated using address generators which do custom addressing for edge and node memories. The generators mainly use the data read for a node as an offset utilized in the addressing for other nodes or edges.

Another study introduces domain-specific accelerator blocks named "Graphicionado" [15], where distinct modules responsible for different operations on the graph are integrated into a pipeline that features several optimizations considering different types of data and memory subsystems. The accelerator pipeline features an on-chip SPM mainly to avoid blocking and wasting the memory bandwidth. Further, it hides the memory access latency by continuously prefetching edge and vertex data with specialized modules. Similar to Graphicionado, "Tesseract" targets the memory bandwidth as a Processing-in-Memory (PIM) accelerator [22]. The architecture features remarkable memory-related optimizations. Using non-blocking message passing, it is able to cover communication latencies. Communication is further optimized with specialized prefetching mechanisms guided by a programming model considering the memory access patterns of graph applications.

The graph analytics accelerators we discussed enjoy sophisticated platforms and report remarkable speed-ups. However, they are generally platform-dependent and not easy to program, either requiring reconfiguration by the designer or an intermediary tool generating the stream. Besides, in these systems where accelerators accompany general-purpose CPUs, additional factors such as the characteristics of the network over which the accelerators communicate with the CPU and the memory hierarchy require attention [23]. These architectures

are difficult to maintain compared to a processor which only needs the instructions generated from a given high-level code of a program.

2.3 Processor-Level Optimizations

Even though many custom graph processors have been discussed, these are mostly accompanied by additional optimizations relying on further advancements in memory systems or accelerators rather than being a processor implementation only. One possible reason might have to do with the new trend of merging general-purpose processors with Field-Programmable Gate Arrays (FPGAs). By the time it was established that processors alone are challenged to meet the needs of graph processing, heterogeneous systems integrating CPUs and FPGAs into the same chip had emerged, attracting the bulk of research interest as they promise both high performance and low cost [21]. Since there is limited work specifically targeting the processor architecture only, our focus is mainly on studies whose contributions on the processor architecture seem the most intriguing.

The oldest attempts try to bring application-specific solutions to graph processing via directly customizing the datapath of the processors for better performance. For example, functions such as graph search [24] and reduction [25] have been optimized. In these architectures, datapaths of general-purpose processors had been extended with additional units such as reconfigurable stack, ALU, and registers used for addressing.

More sophisticated systems emerged in recent years when the FPGA fabric is added to the picture. Zhang et al. developed an FPGA-based graph processor equipped with Hybrid Memory Cube (HMC) [26]. Using HMC, the architecture strives to tackle the memory bottleneck by reducing the memory accesses and optimizing critical parameters shaping the memory traffic. The processor is customized for BFS-like algorithms in a level-synchronized fashion. GraphGen [27] is another FPGA framework which, given a vertex-centric graph specification and

several design parameters provides the developers with application-specific processors automatically compiled for target platforms with accelerators. The tool produces an architecture that includes the RTL descriptions of a processor with separate scratch-pad memories for instructions along with edge and vertex data.

Finally, Song et al. combine a custom graph instruction set with an architecture based on accelerators in a graph processor [8]. Similar to the previous work, they rely heavily on memory system modifications. More specifically, they remove caches from the general-purpose system and include a multi-dimensional toroidal network performing randomized communications corresponding to the footprint of random accesses in graph applications.

The majority of the observations made in the previous section is also valid here. Apart from the first two studies, the systems discussed in this section deliver high performance but they can no longer be considered as processors and they are inherently more difficult to program. Similar to the works discussed under the accelerator section like Tesseract [22], in HMC [26] and Song's work [8], the performance is dependent on advanced memory or networking architectures. Unlike such studies, all of the customization included in our architecture remain in the processor limits as we only alter the processor pipeline, replace cache logic with SPMs, and we do not necessitate any particular main memory or network related features. This facilitates our design to be applied in different systems. Our design is also easier to program compared to frameworks like [27] as our SPMs can be controlled by any programmer through the RISC-V compiler.

Many of the aforementioned studies of Sections 2.2 and 2.3. utilize some common characteristics of graph applications. First, in line with the memory bottleneck problem, the customizations we presented commonly include optimizations on the memory system, interface, or access method. We are thus motivated further to target this bottleneck. Second, some of the studies [22, 27] make use of the separation between edge and vertex traffic in organizing memory units close to the core or while implementing certain functionalities like addressing. This distinction is also utilized in our architecture as we highlight some important differences in the two types of traffic. Finally, noting that the conventional cache

hierarchies are inefficient in graph applications [28], several studies we presented include custom-made SPMs into the design [15, 27]. We take a similar approach by omitting the cache and using separate SPMs for edge, vertex, and global data segments instead.

Chapter 3

Background

In this chapter, we summarize the characteristics of iterative and vertex-centric graph applications, introduce the concept of scratch-pad memories, and give brief backgrounds on the RISC-V ISA as well as the Ibex core.

3.1 Graph Applications

As we consider iterative and vertex-centric graph algorithms in this work, their features are of primary importance. These algorithms process graphs in multiple iterations running until a point that indicates the end of the execution. This point might be marked by a convergence criteria or by the end of a particular task such as searching for a vertex. In vertex-centric algorithms, the operations repeatedly performed throughout the iterations are dictated by the vertices. At every iteration, each active vertex scans its edges and locally applies functions on its own data. This flow of execution is most-commonly referred to as Gather-Apply-Scatter (GAS). While the apply stage corresponds to the local computation performed by the vertex, gather and scatter stages refer to the communication between the vertex and its neighbors. GAS is illustrated in three distinct stages in Figure 3.1.

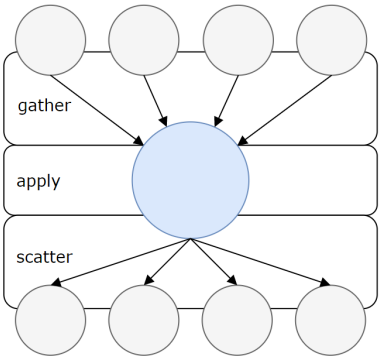

Figure 3.1: Illustration of Gather-Apply-Scatter (GAS).

A well-known algorithm exemplifying GAS is PageRank. In each iteration of PageRank, the vertices gather the ranks of their in-neighbors, perform local computations on the gathered data, and scatter their resulting ranks out to their out-neighbors. The PageRank algorithm can be found in Algorithm 5.1 of Chapter 5 where we examine several graph applications more closely.

The data associated with the neighbors of a vertex in a graph are usually scattered across faraway locations in the main memory. Thus, at each iteration, stages like gather and scatter cause the algorithm to remain poor in terms of spatial locality, resulting in low cache hit rates and the memory bottleneck [11,28]. Besides, since the arithmetic work performed at each iteration is small compared to the number of communications that must be done to reach neighbors, such graph algorithms are considered to have communication-centric workloads [9,10]. This type of workload only contributes more to the memory bottleneck, which we try to address in our work.

3.2 The RISC-V ISA

As the base architecture of our graph processor we chose RISC-V [14], an open instruction set architecture (ISA) introduced in 2010. Unlike many other commercial ISAs, RISC-V is available under free licenses, allowing anyone to design or manufacture RISC-V chips [29,30]. Its other advantages that motivated us are

the following:

- RISC-V has a very rich ecosystem with more than 50 cores and SoCs implemented [31]. Moreover, there is continuous and sufficient software support on several branches like simulators, C compilers, and OS kernels [32]. Free implementations help the specialization process for they provide different cores to advance upon, while the software support makes the hardware testable and usable.

- As a "Base-plus-extension ISA", RISC-V is designed to be extended [30]. It has a small base set separated from several other sets organized aside for standard and non-standard extensions. For instance, Figure 3.2 lists the opcode fields for RISC-V base ISA. For the base set, the first two bits of the instructions, instr[1:0], are "11", whereas the fields instr[4:2] and instr[6:5] are used to classify the instructions into separate opcode spaces. Figure 3.2 indicates which opcode spaces are set aside as *custom* or *reserved* for future extensions. This way, when designers want to customize the ISA with new instructions corresponding to new domain-specific functionalities, they know which fields are guaranteed to remain unused by other sets and they can safely integrate their custom extensions into the architecture.

- RISC-V is proven to be suitable for domain-specific designs. Some areas where custom RISC-V chips with application-specific extensions have been designed include digital signal processing [7], security [33], and isolated execution [34].

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Figure 3.2: RISC-V base opcode map with fields reserved for custom extensions [29].

3.3 The Ibex Core

Ibex is an open 32-bit RISC-V core which can freely be used and modified [35,36]. It has a 2-stage in-order pipeline and it supports RV32I, RISC-V Base Integer Instruction Set, as well as C (compressed), M (integer multiplication and division), B (bit manipulation), Zicsr (control and status register), and Zifencei (instruction-fetch fence) extensions. The pipeline, displayed on Figure 3.3 taken from the Ibex User Manual [37], consists of an IF (instruction-fetch) stage separated from an ID (instruction-decode) stage accompanied by an execute block and a load-store unit. We preferred Ibex to constitute the base of our graph processor because it is a relatively small core that can easily be customized. It also does not include advanced architectural features, which allows us to keep our work independent from any special hardware level optimization.

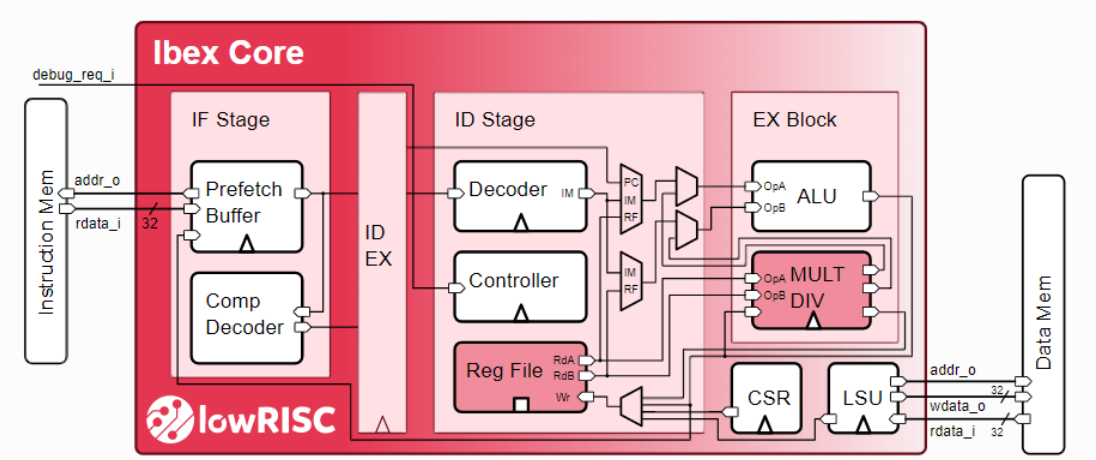

Figure 3.3: The block diagram of the Ibex Core [37].

3.4 Scratch-Pad Memory (SPM)

Scratch-pad memories (SPMs) are fast on-chip memories that, unlike hardware-controlled caches, are controlled by the software [38,39]. Either the application or the compiler should manage the data traffic of an SPM with explicit load, store, or erase mechanisms. Via SPMs, the efficiency and the performance of architectures can potentially increase because the complex tagging logic of caches is eliminated and the on-chip area is filled with more useful data, especially if the software is able to follow the execution order well enough to know when to schedule which data for the SPM. Similar to the previous work [15,27] discussed in Chapter 2, we integrate SPMs into our design. Considering the unique access patterns of graph applications, we hope to utilize the software's intelligence to better organize the memory traffic and to reduce the number of accesses that cause the pipeline to get stalled.

Chapter 4

The Architecture

Before introducing the proposed graph processor architecture in detail, we start by pointing out the motivation behind designing different types of SPMs. The main memory accesses in graph applications can be roughly categorized into two types: edge traffic and vertex traffic. The vertex being processed first should lookout to see who its neighbors are. Then, its data contributes to or is contributed by the data of these neighboring vertices. The topological information that needs to be collected in the first part of this typical execution pattern accounts for the edge traffic. It enjoys high spatial locality as the edges of a vertex are usually stored contiguously and it is read-only because the topology of the graph is fixed. After the edges are discovered, the execution needs to go through a vertex traffic where the data of the vertices being involved is fetched or updated. Since the neighbors of a vertex are usually spread across the graph, their data is stored away from each other, and reaching out to these remote locations requires irregular memory accesses with very poor locality. The presence of these two distinct types of traffic seem to be a ubiquitous feature in all graph applications.

Motivated by this observation, we introduce different scratch-pad memories (SPMs) to reduce the edge and vertex traffic independently. For edge data, we designed the **Edge Scratch-Pad** (ESP). ESP is a strictly software-controlled read-only memory whose efficiency depends on how predictable the processing

order is. For vertex data, we provide **Vertex Scratch-Pad** (VSP) which is a read/write memory that is allocated only for a certain portion of vertex data, also referred to as *VSP Data* throughout the text. The software is responsible for configuring VSP such that it maps to the memory range corresponding to the vertex data that makes the heaviest contribution to the memory load of the application. To handle the vertex data that remains outside the range of VSP, the edge data that does not fit into or cannot be scheduled for ESP, and any other sort of data left aside, we include the cache-like **Global Scratch-Pad** (GSP) in the design.

In essence, what is presented is a processor where the closest memory units are custom-made for graph applications such that, when accurately organized and controlled by the software, they enjoy smaller numbers of blocking memory accesses thanks to reduced miss rates. Being a processor, the architecture comes with both the ease of programmability and the flexibility in operating modes: The software can prefer to use the hardware as a general-purpose processor simply by configuring SPMs as conventional caches mapping to whole memory.

To configure, load, read from, and erase these SPMs, we extend RISC-V ISA with `spmcon`, `memspm`, `spmreg`, and `delspm` instructions, respectively. Implementation of these instructions is given in this chapter, while some examples demonstrating how these instructions are used by the software can be found in Chapter 5.

In the following sections, we describe the architectural details of ESP, VSP together with GSP, and the final design where the three SPMs are combined.

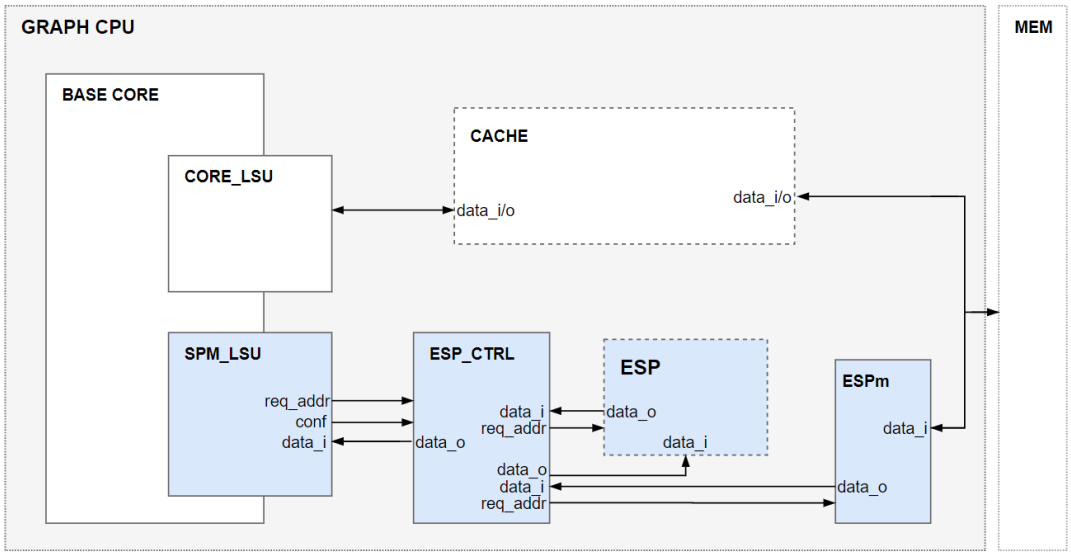

Figure 4.1: The diagram of the architecture with ESP.

4.1 Edge Scratch-Pad (ESP)

4.1.1 Overview

The Edge Scratch-Pad, ESP, is responsible for the edge data. It is configured, loaded, read, and erased by the software using the custom instructions described in Chapter 5. During the application, the edge data associated with the vertices who are likely to be processed soon are loaded into ESP so that when it is time to process these vertices, at least some of their neighbors had been prefetched and are already present in ESP. The preload occurs in parallel with datapath's execution so that the pipeline is not stalled.

Figure 4.1 shows the graph processor equipped only with ESP. The schematic emphasizes the data flow along with requests and configuration signals. ESP merely carries some of the edge data so it must be accompanied by a cache. It connects to the datapath via the unit called SPM_LSU, (the Scratch-Pad Memory Load-Store Unit) and is controlled by ESP Controller. Upon receiving the operation type and target address from SPM_LSU at once, the controller proceeds to use ESP by itself. On the memory side, if the controller understands that it is

time to load ESP, a sub-unit called ESPm makes the memory request and fetches the data for ESP Controller, which then either writes the data onto ESP or saves it to be used later.

In an architecture similar to Figure 4.1 the presence of ESP is expected to decrease the number of datapath-blocking memory accesses made by the cache in two ways. First, each read request that hits in ESP means that the core issues one less load-word instruction. Having found the data it is looking for in ESP, the software refrains from making any further requests on the same address. This should decrease the compulsory misses that might otherwise occur in the cache at a certain rate. Second, in such a scenario the traffic on the cache alone is decreased because ESP takes some of the load, reducing the amount of conflicts in the cache. We will discuss the units involved in the operation of this SPM starting from its controller.

4.1.2 ESP Controller

The ESP Controller, also referred to as ESP_CTRL, is the main module where the ESP functionality is implemented. It introduces inherent parallelism into the processor: After receiving an SPM related instruction from SPM_LSU, the controller single-handedly manages ESP, allowing datapath to continue its execution. ESP is loaded and erased in chunks, and read word by word. *Chunksize*, echoing the block size of a cache, is the parameter corresponding to the number of words with which load and erase mechanisms work.

4.1.2.1 Implementation of `spmcon`

To configure ESP, the software informs the ESP Controller about the starting address of the edges of the graph structure by sending the address with `spmcon` instruction. This memory address is later used as a base for `memspm`. It is indexed with offsets to reach the memory location where the neighbors of the node with the given offset start.

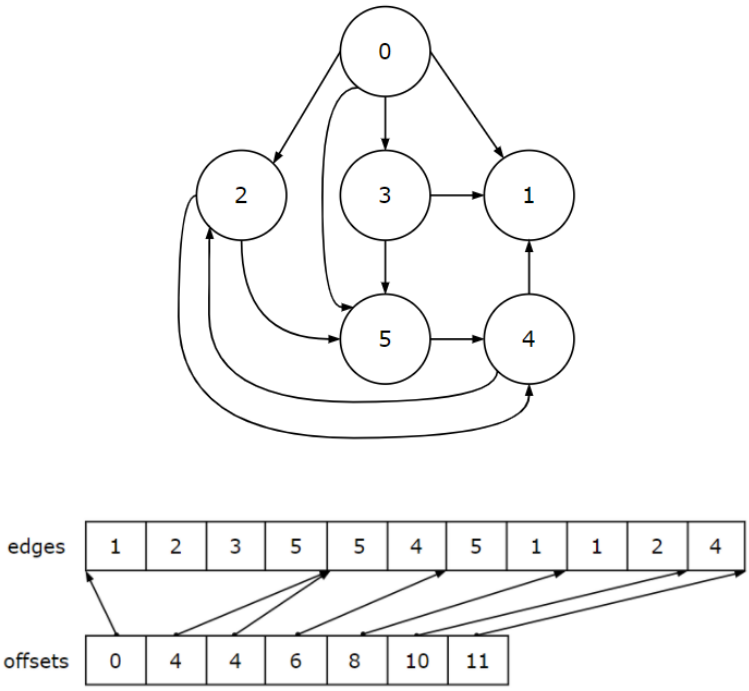

Figure 4.2: Example graph in CSR representation.

This implementation requires the graph to be stored in Compressed Sparse Row (CSR) format [40]. For instance, in an example graph with 6 nodes, 11 edges, and the CSR representation in Figure 4.2, ESP must be configured by the software sending the address *edges* to ESP Controller using `spmcon` instruction.

4.1.2.2 Implementation of `memspm`

With `memspm`, a custom addressing mechanism that fills ESP with up to *chunksize* edges of any vertex is initiated. Sending the offset of the vertex to be loaded with `memspm`, the processor assigns the controller with the task of calculating the target address and performing the fetch. Figure 4.3 and the following step-by-step procedure describe how the controller uses the address sent with `memspm` to load the edges of an arbitrary vertex i into ESP.

1. The software sends the address *offsets+i* with the `memspm` instruction

   (a) The address is stored in a FIFO buffer until a chunk in ESP is free. ESP is freed with `delspm` instruction that is presented in the next subsection.

2. The controller reads the values stored at addresses *offsets+i* and *offsets+i+1*, which are kept in *off1* and *off2*.

   (a) If *off1* = *off2*, vertex i has no edges and ESP is not loaded.

   (b) If *off1* != *off2*, vertex i has edges and they are found in a range whose starting address can be calculated using the offset kept in *off1*.

3. The controller calculates the address of the first edge of vertex i as *edgead* = *edges* + *ptr1*. *edges* is the base address sent with `spmcon` earlier during the configuration.

4. Starting at *edgead*, the controller loads *chunksize* successive words into ESP and saves their addresses on an address table for future reads. The chunk loaded now includes at least some of the edges of the vertex i.

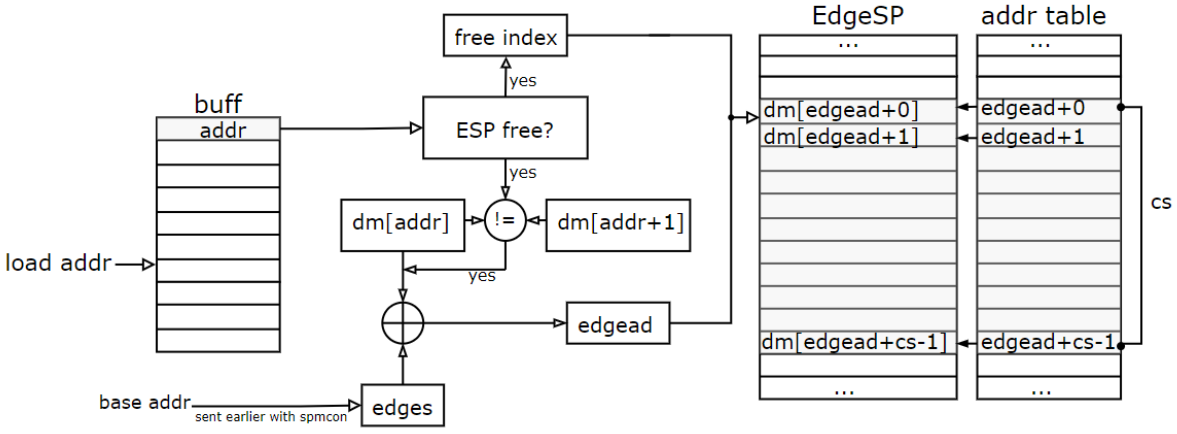

Figure 4.3: Loading of ESP, initiated by `memspm` instruction.

Such an implementation is practical in several ways. First, the software knows that ESP contains the data of a node only if the node has any. This establishes an agreement between the software and the hardware on when to use `delspm` instruction. Second, *chunksize* words are written into the SPM in an automatic manner initiated merely by an address. The software states which vertex it needs and ESP is filled without requiring the software to provide any further addresses. Finally, an SPM to be filled with edge data is convenient for the memory interface because of the high locality and read-only nature of the data. To retrieve *chunksize* words, ESP only needs to make an average of *chunksize*/*bandwidth* requests, rendering the communication bandwidth-efficient. The communication is also easy to implement and coherence-friendly as it is read-only.

One obvious drawback of loading ESP this way is the predictability challenge. As illustrated in Chapter 5, ESP functionality is induced into algorithms in dramatically different manners. In a typical PageRank implementation [41] for instance, predictability is very high because the order with which the vertices contribute to their neighbors is known, whereas in Dijkstra's SSSP [42], when a vertex is found to have the second minimum distance on an iteration, it should promote to be the first minimum on the next iteration for the prediction made previously to be immediately useful. Furthermore, one possible predictor is the active status of the nodes, but in many applications, the active list tends to grow too rapidly for the FIFO buffer in Figure 4.3 to accommodate. This is apparent for an application such as BFS where the majority of the vertices are discovered in the first few iterations [41]. ESP is read out much slower than it is attempted to be filled and there are too many vertices stranded on the new arrival buffer waiting to be loaded, introducing its capacity as a potential bottleneck.

4.1.2.3 Implementation of `spmreg` and `delspm`

In the implementation of `spmreg`, the address sent along is first searched for in ESP address table. ESP itself and its address table, which was also populated as ESP was being loaded, are connected such that the ith entry on the address table corresponds to the address of the data sitting at the ith entry of ESP. When the

controller receives an address sent along with `spmreg` instruction, it searches for a match in the address table. If there is a match, the index is noted and the data stored on the same index is fetched from ESP. If the address is not found, the controller returns a special reserved value indicating that the request cannot be served. The software should then infer that it has to make a memory request. This mechanism allows the software to save on ESP requests as well. When a request related to a vertex is missed on ESP, the software might infer that it should not make any further requests for the current vertex.

A similar procedure involving the search for a match in the address table is carried out for `delspm` too. This time the address sent should be the address of the first neighbor of the node that has just been processed and the entire chunk associated with the node is deleted at once. Implementation of such a function requires the indices of ESP to gather in chunks so that each chunk can be indexed separately. An illustration of chunk indexing for an ESP with 4 chunks and 4 words per chunk is given in Figure 4.4.

In Figure 4.4, there are 4 chunks and thus 4 entries on the free chunk table. When the software sends the address of the first entry of a chunk with `delspm`, the controller first searches for the address in the table shown in the first row of Figure 4.4. If the address sent with the instruction matches with any address on the table, the index of the matching entry of the table is noted. Next, the controller uses the matched index to calculate the corresponding chunk index via

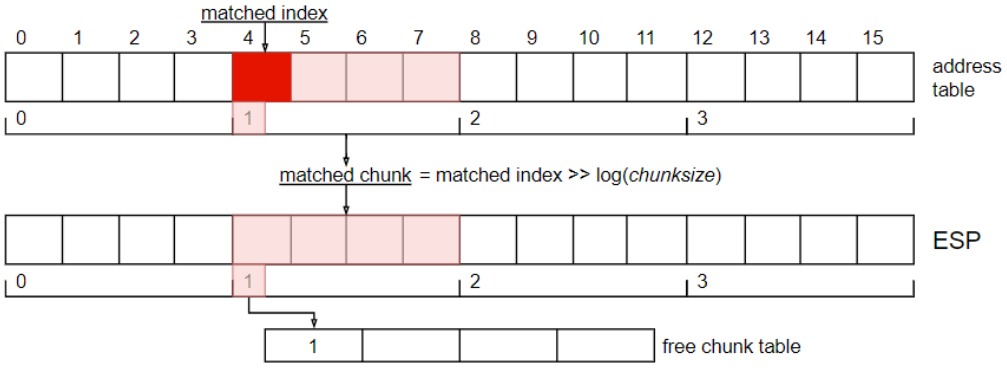

Figure 4.4: Chunk indexing in `delspm` instruction.

simple shift and logarithmic operations. This chunk is then deleted, or marked as free to be overwritten. Its index is put in the first available location of the free chunk table, which is controlled by a simple counter. The free chunk table is later used by `memspm` routine at the stage of deciding where in ESP to put the data to be loaded. Once a previously free chunk has been reloaded, the free chunk table is shifted and the associated counter is decremented.

4.2 Vertex Scratch-Pad (VSP) and Global Scratch-Pad (GSP)

In Section 5.1.1, we mentioned that ESP needs to be accompanied by a cache to handle accesses related to the rest of the memory traffic. What we try to achieve with VSP and GSP is to customize a general-purpose structure like a cache in a way that is more suitable for the memory access patterns of graph applications. VSP takes on the traffic stemming from the portion of vertex data that presents the heaviest load on the application. We refer to this portion of vertex data, or this range in the memory as the *VSP Data* as well. We discuss the motivation behind separating VSP Data from the rest of the vertex traffic in Chapter 5.

When a load/store on the VSP Data is executed, related data is brought in to VSP which exclusively maps to the range described by the limits of the VSP Data in memory. GSP takes the role of a cache for the memory that remains outside of the range of the VSP Data, handling the rest of the accesses including the edge traffic and the vertex traffic not associated with the VSP Data. This distinction is illustrated on a hypothetical memory array in Figure 4.5.

Figure 4.6 gives the modified processor architecture utilizing the VSP and GSP. There is a common controller called the Scratch-Pad Controller or **SP_CTRL** that communicates data and configuration parameters to LSU units. The structure can make one memory request at a time, which is either from VSP or GSP and it is categorized by an arbiter referred to as the Scratch-Pad Arbiter or

SP_ARB. Both VSP and GSP are adapted from the cache design provided in the literature [43]. It is therefore trivial for the software to configure GSP as a cache. Assigning VSP with 0 bytes, the software directly switches into the general-purpose mode with GSP serving to the entire memory.

VSP and GSP are accessed with standard load/store instructions, and configured by the custom `spmcon` instruction which was also used to configure ESP. While `spmcon` flows through SPM_LSU, the loads and stores are handled by the Core Load-Store Unit that is used to make standard memory requests by the core. To configure these units, the software should send two `spmcon` instructions carrying the start and end addresses of the range that contains the VSP Data. For instance, when the software decides that an array called *front* is to be set aside for VSP, it first should send the address *front* to ESP Controller. Being a part of the vertex data, this array has the same size as the number of vertices (denoted by N) so the software completes the configuration by sending the address computed by adding N-1 to *front*, which marks the end of the range for VSP. Then, SP_CTRL and SP_ARB, make sure that the requests within the range of VSP Data are directed correctly. In our example, the requests related to the elements of the vector *front* are directed to VSP and those outside the range are directed to GSP.

The partnership of VSP and GSP is mostly expected to decrease the conflict misses. As will be discussed in the next chapter, the VSP Data is the source of an important percentage of memory accesses, and when it maps to the same memory unit as other data (as in the case of a conventional cache), it might kick out the blocks that would otherwise be useful with their spatial or temporal locality. In a scenario where we have VSP and GSP instead of one compact cache, for instance, the edge traffic should enjoy more hits stemming from better utilized spatial locality because the conflicts which would normally be led by the VSP Data are fully eliminated. VSP not only eases the load of GSP but also offers a private space for the VSP Data, which should therefore experience less compulsory misses.

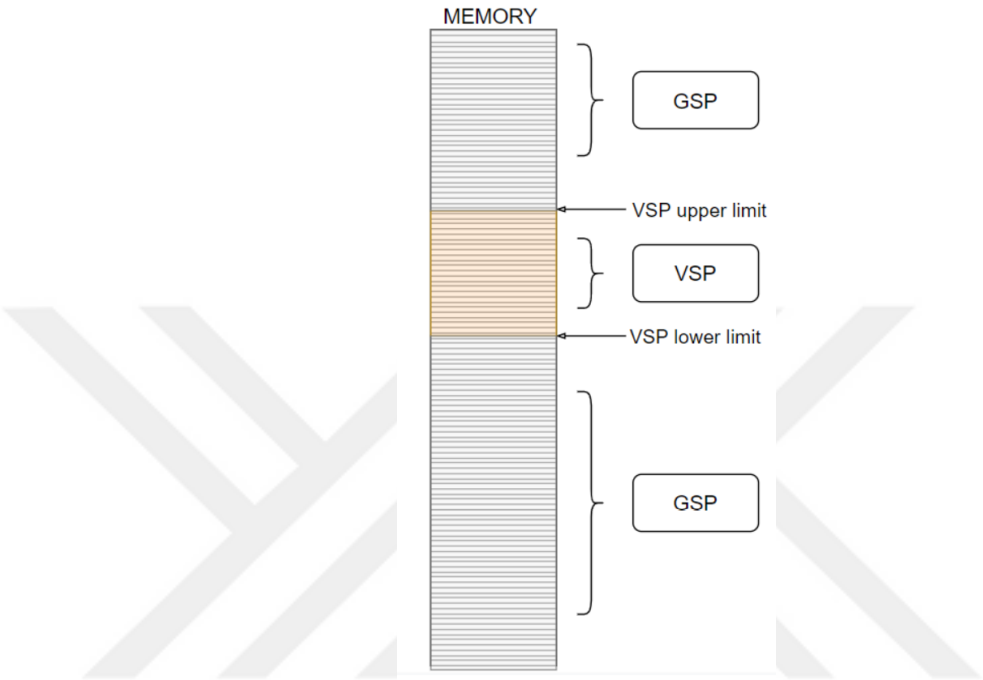

Figure 4.5: Different portions of the data memory are configured to map to either VSP or GSP.

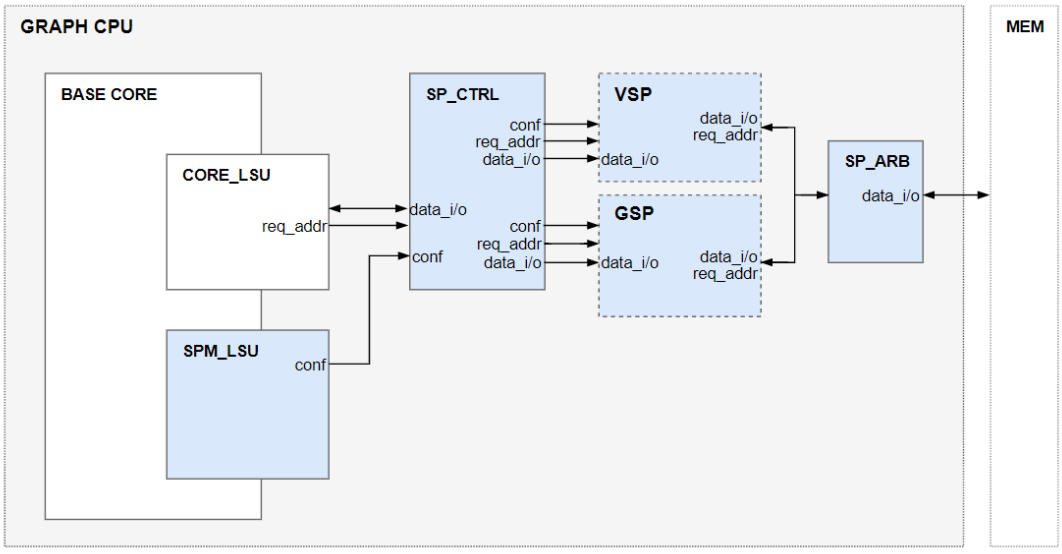

Figure 4.6: The diagram of the architecture with VSP and GSP.

4.3 Advanced Architecture

Figure 4.7 presents the advanced architecture for the graph processor where ESP, VSP, and GSP together replace the cache. The related instructions sent by the software are first forwarded to Core and SPM Load-Store Units which connect to the controllers on the hardware side. The flow of instructions can be observed in Figure 5.3 of the next chapter. Exclusive controllers that manage ESP, VSP, and GSP are preserved while the SPM_LSU is extended to combine different functionalities needed to integrate all three types of SPMs. It configures ESP by providing ESP_CTRL with the base address of the edges, and VSP by providing SP_CTRL with the lower and upper addresses marking the borders of VSP Data. It also communicates with ESP Controller to exchange instructions and data. The Core Load-Store Unit continues to carry the data associated with VSP and GSP, while SP_CTRL and SP_ARB guarantee that the memory requests are correctly categorized between the two.

In the next section, we complete the discussion on the design of the graph processor by formally introducing our ISA-level extension and by describing the software aspect with example applications.

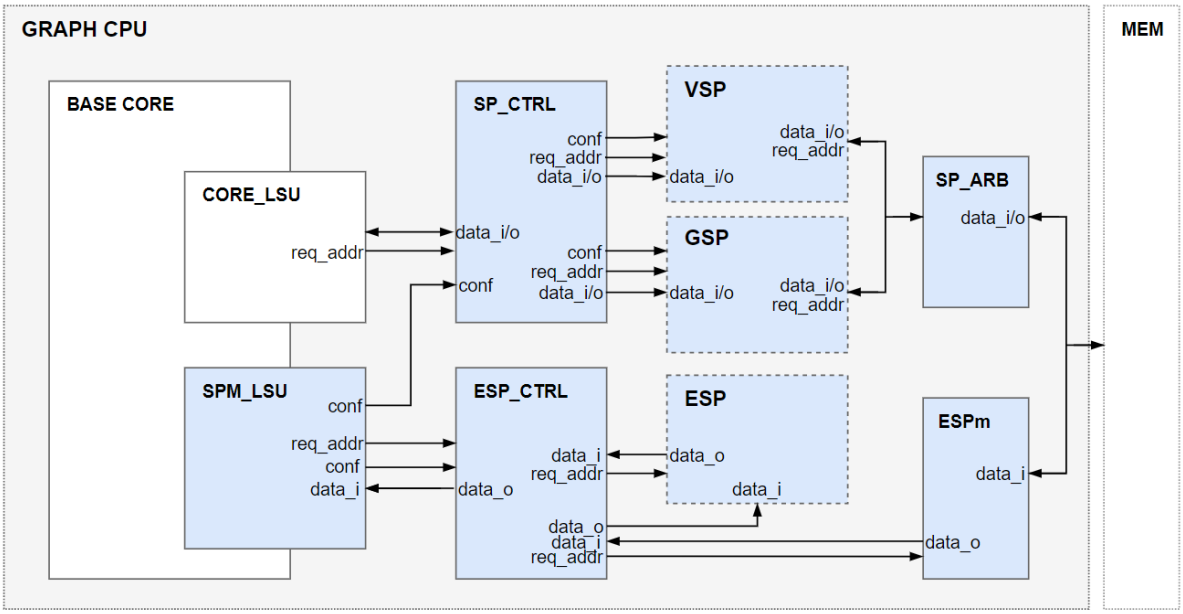

Figure 4.7: The graph processor architecture with ESP, VSP, and GSP.

Chapter 5

Custom Instructions and the Software

In this chapter, we first introduce the ISA-level support for the graph processor by defining the custom instructions added to the RISC-V ISA. We then visit PageRank (PR), Single-Source Shortest Path (SSSP), and Breadth-First Search (BFS) algorithms to illustrate how graph applications should be rewritten for the graph processor. We conclude the discussion on the design by giving motivation on offering an exclusive traffic for the VSP Data and by illustrating the general flow of execution.

5.1 Custom Instructions

We propose to extend RISC-V ISA with `spmcon`, `memspm`, `spmreg`, and `delspm` instructions to configure, load, read, and delete data from the SPMs introduced in Chapter 4. These instructions are the tool with which the software running on the graph processor controls the SPMs. In accordance with RISC-V extension standards, we define the instructions on the opcode field instr[6:0]=0001011 reserved for custom extensions.

We rebuilt GCC RISC-V compiler version rv32ima [44] with these new instructions added into the ISA files, enabling the compiler to produce the machine code of the new instructions when they are included in the high-level program as in-line assembly segments. We wrap the segments in functions whose arguments correspond to operands. The instructions are listed below with their example use, functional definition, target SPM, and encoding:

- Configure SPM: spmcon rd, rs1, rs2
  - Example use : spmcon(edge,0)
  - Function : Configuration of SPMs by providing their controllers with base addresses.
  - Target SPM : ESP, VSP, GSP
  - Encoding: `spmcon rd rs1 rs2 31..25=0 14..12=0x1 6..2=0x02 1..0=3`

- Memory to ESP: memspm rd, rs1, rs2
  - Example use : memspm(offset,0)
  - Function : Non-blocking data transfer from memory to ESP Controller. If a load is to be performed, *chunksize* successive words starting at the memory address *edge+offset* are fetched for ESP.
  - Target SPM : ESP
  - Encoding : `memspm rd rs1 rs2 31..25=0 14..12=0x4 6..2=0x02 1..0=3`

- ESP to Regfile: spmreg rd, off(rs1)
  - Example use : data = spmreg(addr)
  - Function : Data transfer from ESP to register file. Reads the word whose memory address is *addr* into the register referred to as *data*. *data* will contain a reserved value if the word is not found in ESP.

  - Target SPM : ESP
  - Encoding : `spmreg rd rs1 imm12 14..12=0 6..2=0x02 1..0=3`

- Delete from ESP: delspm rd, rs1, rs2
  - Example use : delspm(addr1, addr2)
  - Function : Marks the ESP chunk whose starting word lies on the memory location indicated by *addr1+addr2*. The chunk can now be overwritten by new data.
  - Target SPM : ESP
  - Encoding : `delspm rd rs1 rs2 31..25=0 14..12=0x2 6..2=0x02 1..0=3`

`spmcon`, `memspm`, and `delspm` instructions only compute the addresses that will be forwarded to SPM units. We therefore define them as R-type instructions. For the implementation to remain convenient, we keep the field instr[31:25] the same. We utilize the funct fields given in instr[14:12] to differentiate between the three instructions. On the hardware implementation, the identification based on instr[14:12] is done in the decode stage. Upon understanding an SPM-related instruction has arrived on the datapath, the decoder configures the execute block to perform addition as well. The values in the operand registers of R-type SPM instructions are added together to compute the address to be sent to SPM controllers. `spmreg` is defined according to the RISC-V I-type standards for it imitates the regular I-type load-word instruction. All instructions have the same opcode field embodied by the bits instr[6:0], which puts them under the same custom extension set we defined.

5.2 Applications on the Graph Processor

For this work we consider PR, SSSP, and BFS algorithms. In all of the applications, the rationale behind calling `spmcon`, `spmreg`, and `delspm` will be similar.

`spmcon` must be used three times at the beginning of the execution. The first call sends the address *edges* for ESP, and the following two calls send the addresses of the first and last words included in the VSP Data. `spmreg` is used whenever the software requests to read a word from ESP. The applications are modified in a simple manner which can easily be mimicked for other applications. Whenever the software reaches the point where it needs to discover the neighbors of a vertex, it first makes a request at ESP, assuming it will contain the edges. If the result is a hit, the data is returned and the software continues to use `spmreg` for the current vertex. When the request is missed, it either means the vertex has more edges than *chunksize* or was not put into ESP in the first place. In both cases, the software does not apply `spmreg` anymore for the current vertex and returns to making memory accesses.

`delspm` is called for every vertex with a nonzero number of outgoing neighbors. This is because on the hardware side, ESP Controller first checks whether a vertex has any neighbors before attempting to load them into the ESP, as mentioned in Chapter 4. This feature automatically renders `delspm` for vertices that have no neighbors useless, giving the software a simple way to decide for which vertices to use the instruction. As for the timing, we should call `delspm` for a vertex right after we finished processing it.

The inclusion of `memspm` is different in each application. It is used at the points where the software can guess which vertices will be processed in the near future. This predictability factor proves to change depending on the application. Thus, we examine the use of `memspm` independently within separate sections dedicated to each application.

In the coming sections, we describe the basic algorithms as benchmarks for future reference and we demonstrate how to update them to use the graph processor. We considered the most conventional versions of the algorithms so that our work will remain independent of any specific optimization and can give a general idea of both the use and the performance of the graph processor.

5.2.1 PageRank (PR)

Pagerank (PR) is a well-known algorithm used by search engines to rank pages based on their popularity [3]. In this computation, the popularity or the pagerank of any vertex is scaled by its number of outgoing links to contribute to the pageranks of the vertices it links to. This interaction can also be explained in terms of the Gather-Apply-Scatter (GAS) steps where each independently modeled vertex gathers the pagerank values of its incoming neighbors, applies a short arithmetic function to the values it received to calculate its own pagerank and scatters the resulting value out to its out-neighbors [45]. Since this pattern results in dependencies between iterations, a threshold that marks the convergence point is employed to stop the iterative execution.

We give the classical push-based vertex-centric PR in Algorithm 5.1. The alpha value used to control the convergence is set to 0.001 as it is widely adopted in the literature [46]. We stop when the sum of the absolute values of the difference between the reciprocal elements of *newRanks* and *Ranks* are smaller than alpha, another popular choice indicating the convergence.

Algorithm 5.1 Conventional PR

```
Input: G = (V,E)
Output: ranks
    ranks[:] = 1/nodeNum
    newRanks[:] = 0
    for i until maxIter do
        for u ∈ V do
            rank=ranks[u]/outNeighborNum(u)
            for v ∈ outNeighbor(u) do
                newRanks[v] = newRanks[v] + rank
            end for
        end for
        for u ∈ V do
            ranks[u] = base + d*newRanks[u]
        end for
        newRanks[:] = 0
    end for
```

Algorithm 5.2 SPM-Aware PR

```
Input: G = (V,E)
Output: ranks
    spmcon(edges)
    spmcon(newRanks)
    spmcon(newRanks+nodeNum)
    ranks[:] = 1/nodeNum
    newRanks[:] = 0
    for i until maxIter do
        for u ∈ V do
            memspm(u+off)
            rank=ranks[u]/outNeighborNum(u)
            for v ∈ spmreg(u) or ∈ outNeighbor(u) do
                newRanks[v] = newRanks[v] + rank
            end for
            if outNeighborNum(u) ≠ 0 then
                delspm(u)
            end if
        end for
        for u ∈ V do
            ranks[u] = base + d*newRanks[u]
        end for
        newRanks[:] = 0
    end for
```

In Algorithm 5.2 we modify the Conventional PR to make use of the additional graph processor functionalities. Specifically, in the modified version with SPMs, the software initially configures VSP to map to the array *newRanks* which holds the most current pagerank values repeatedly calculated at each iteration. Then, in each loop, it uses `memspm` to load the vertex *off* ahead of the current vertex. For example, if *off* is 8, at the instance when ranks[2] (the vertex with ID 2) is processed, the outgoing edges of the vertex with ID 10 are scheduled to get loaded to ESP. Thus, some of the outNeighbors(10) will already be present there after 8 turns.

`spmreg` and `delspm` follow the logic explained at the beginning of this chapter. For each outgoing edge of any vertex u, ESP is checked first. The main memory is accessed only when the SPM request is a miss. `delspm` is called for vertex u if it had any outgoing edges and right after u's turn has been completed - resulting

in its edge data getting erased from ESP.

5.2.2 Single-Source Shortest Path (SSSP)

Given a source vertex, single-source shortest path (SSSP) algorithms return the shortest paths to all reachable vertices in the graph. We consider the well-known Dijkstra's algorithm solving the SSSP problem in Algorithm 5.3 [47, 48].

Requiring that the edge weights are positive, Dijkstra's algorithm is a greedy algorithm where the vertex which has the smallest distance to source is selected at each iteration. The selected vertex then updates the distances of its neighbors. In our implementation, whether a vertex has got its final distance or not is controlled by an array named *set*. We iterate the algorithm until the distance values stop changing, which is inferred when the value returned in *next* remains the same for two consecutive iterations.

Algorithm 5.3 Conventional SSSP

```
Input: G = (V,E),source, w
Output: dists,parents
    dists[:] = +∞
    set[:] = false
    dists[source] = 0
    set[source] = true
    for i until maxIter do
        next = exractMin(set,dists)
        set[next] = 1
        for v ∈ outNeighbor(next) do
            if set[v] ≠ 0 then
                if dists[v] > dists[next] + w(next,v) then
                    dists[v] = dists[next] + w(next,v)
                    parent[v]=next
                end if
            end if
        end for
    end for
```

Our modified version of the SSSP given in Algorithm 5.4 utilizes the available SPMs. We allocate the array *set* for the VSP as it is the most often visited structure in the algorithm, needed both when extracting the closest vertex and before updating the distances. While extracting the vertex with the minimum distance, we first check whether the vertex being considered is set before retrieving its distance. The same applies when updating the distance values of the remaining vertices: Those who have already been set should not even have their distance values checked. We use `spmreg` and `delspm` in a similar fashion as in PR.

Algorithm 5.4 SPM-Aware SSSP

```
Input: G = (V,E),source, w
Output: dists,parents
    spmcon(edges)
    spmcon(set)
    spmcon(set+nodeNum)
    dists[:] = +∞
    set[:] = false
    dists[source] = 0
    set[source] = true
    for i until maxIter do
        next, secNext = exractMin(set,dists)
        memspm(secNext)
        set[next] = 1
        for v ∈ spmreg(next) or ∈ outNeighbor(next) do
            if set[v] ≠ 0 then
                if dists[v] > dists[next] + w(next,v) then
                    dists[v] = dists[next] + w(next,v)
                    parent[v]=next
                end if
            end if
            if outNeighborNum(next) ≠ 0 then
                delspm(next)
            end if
        end for
    end for
```

To infer which vertices might be needed soon, extractMin function is arranged such that it not only returns the closest but also the second closest remaining vertex as well. After this, the software triggers the hardware to load this second closest vertex into ESP by issuing the `memspm` instruction.

5.2.3 Breadth-First Search (BFS)

Breadth-First Search (BFS) is another popular application that finds the breadth-first order traversal of a graph starting at a source vertex. The most conventional BFS code with top-down pushes is presented in Algorithm 5.5 [41, 42]. In each iteration, the active vertices are put into the *frontier* array which controls whether a vertex has been discovered or not. Different implementations might include a coloring mechanism as well as other structures utilized to keep track of some other properties of the vertices such as their breadth-levels and distances to the source [41].

Algorithm 5.5 Conventional BFS

```
Input: G = (V,E), source
Output: parents
    frontier ← source
    next ← {}
    parents[:] = -1
    while frontier ≠ {} do
        for v ∈ frontier do
            for u ∈ outNeighbor(v) do
                if parents[u] = -1 then
                    parents[u] = v
                    next ← next U u
                end if
            end for
        end for
        frontier ← next
        next ← {}
    end while
```

The modified version to be executed on the graph processor configures the SPMs as the first step similar to the previous algorithms. As can be seen in Algorithm 5.6, this time VSP is assigned for *frontier*, the array that keeps the activated vertices at each iteration. For each vertex, we initially assume ESP will contain the adjacency data and we erase any vertex that has outgoing edges right after we finished processing it, similar to the previous two applications.

Algorithm 5.6 SPM-Aware BFS

```
Input: G = (V,E), source
Output: parents
    spmcon(edges)
    spmcon(frontier)
    spmcon(frontier+nodeNum)
    frontier ← source
    next ← {}
    parents[:] = -1
    while frontier ≠ {} do
        for v ∈ frontier do
            for u ∈ spmreg(v) or ∈ outNeighbor(v) do
                if parents[u] = -1 then
                    parents[u] = v
                    next ← next U u
                end if
            end for
            if outNeighborNum(v) ≠ 0 then
                delspm(v)
            end if
        end for
        memspm(next[:])
        frontier ← next
        next ← {}
    end while
```

The most important hint the software can use to guess which vertices will be processed in the following iterations is the array *next*. *next* contains the vertices on the next breadth-level, directly dictating the order of processing for the upcoming iteration. Therefore, we load the edges of the vertices in *next* into the ESP as much as its capacity permits. We give the modified BFS in Algorithm 5.6.

5.3 Motivation on VSP Data

Having examined some of the most popular graph benchmarks, we are able to observe the distinction between the VSP Data and other types of data which might also be considered within the vertex traffic more clearly.

In PR, *newRanks* get $|E|$ requests, whereas *ranks* get only $|V|$ during the gather/scatter stage of each iteration. This implies that the array *newRank* should introduce a much heavier load on the memory traffic, and it is thus best to allocate a separate space for it. Similarly, in BFS the choice for VSP is the vector *frontier* because it is needed each time the main for loop iterates, and updated right after the loop on each iteration. Finally, in SSSP the array *set* is exhaustingly more often visited than any other array keeping vertex data.

In Figure 5.1, we demonstrate the number of requests VSP and GSP receive for PR, SSSP, and BFS running on a sample graph (G3, as will be explained in Chapter 6). The architecture under test is given in Figure 4.6. Without ESP, the logic behind the separation of GSP and VSP can be observed more accurately.

The observation of VSP taking around half of the total memory requests highlights how frequently the VSP Data is accessed for traversal or update purposes in these applications. We are thus motivated to direct the traffic associated with the VSP Data to a space separate from others. Unlike vertex data, edge data should usually enjoy high locality, which is expected to be manifested in an architecture where edge data is isolated from at least some of the vertex data-led conflicts.

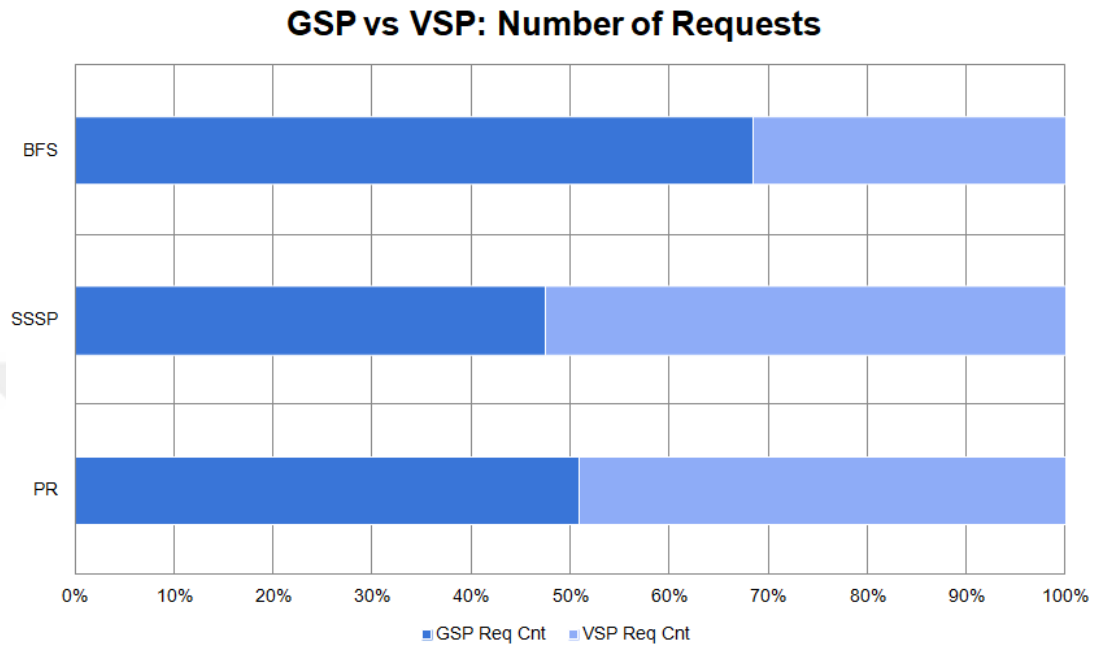

Figure 5.1: Percentage of memory requests directed at VSP and GSP.

We note that, in different applications or in different versions of the applications we consider, it might intuitively be more reasonable to allocate VSP for other memory blocks. Until a formal guideline for each application is established, making the most intelligent decision will be up to the software and as long as it remains intelligent, which is what we usually expect from the software side, the benefits of VSP should continue to be apparent.

5.4 Illustration of the Framework

Before we conclude this chapter, we would like to present a summary of our complete framework. We use an excerpt from PR given in Algorithm 5.2 to illustrate how the software interacts with the hardware and how the custom instructions are actually implemented.

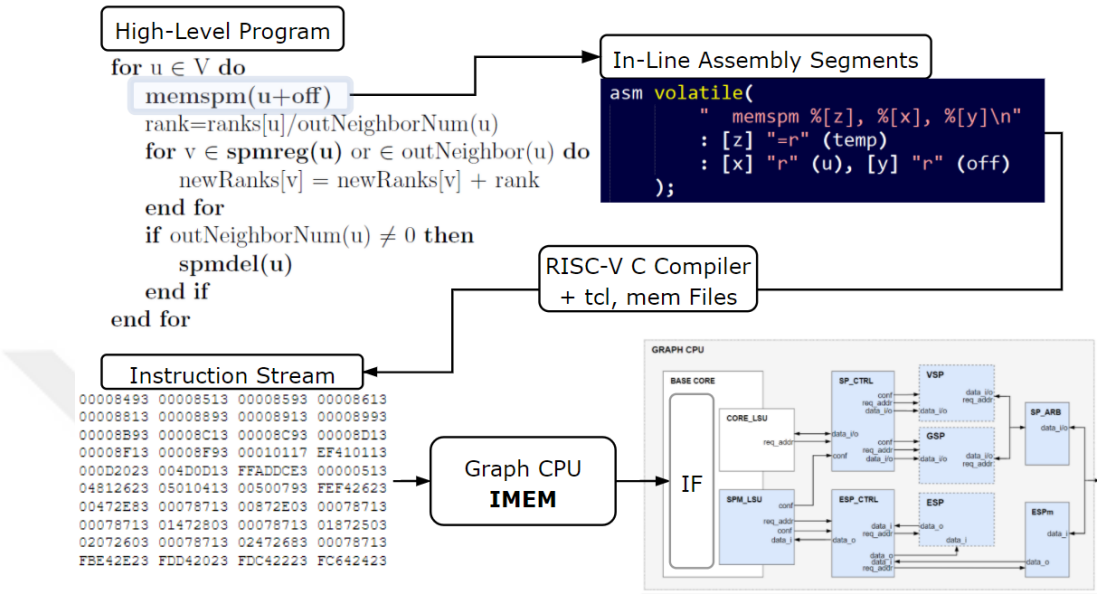

Figure 5.2: Software-hardware interface for the custom instructions.

Figure 5.2 explains the generation of custom instructions. We start with a C implementation of the algorithm given in Algorithm 5.2. Each highlighted line corresponds to a function featuring the actual custom instructions in in-line assembly form. After recognizing the instructions in in-line assembly, the compiler arranges the registers by assigning them the values held in the argument variables and it produces the binary code for the instruction. Then, just like other instructions, custom instructions like `memspm` are put to their optimal places in the compiler generated assembly file. Next, the compiler uses these binaries and some additional scripts initializing the registers and the memory to generate an executable stream of instructions. Finally, the stream is loaded to the graph processor's instruction memory (IMEM) which connects to the core through the fetcher inside its IF stage. The program running the PR can thus be tested on the graph processor.

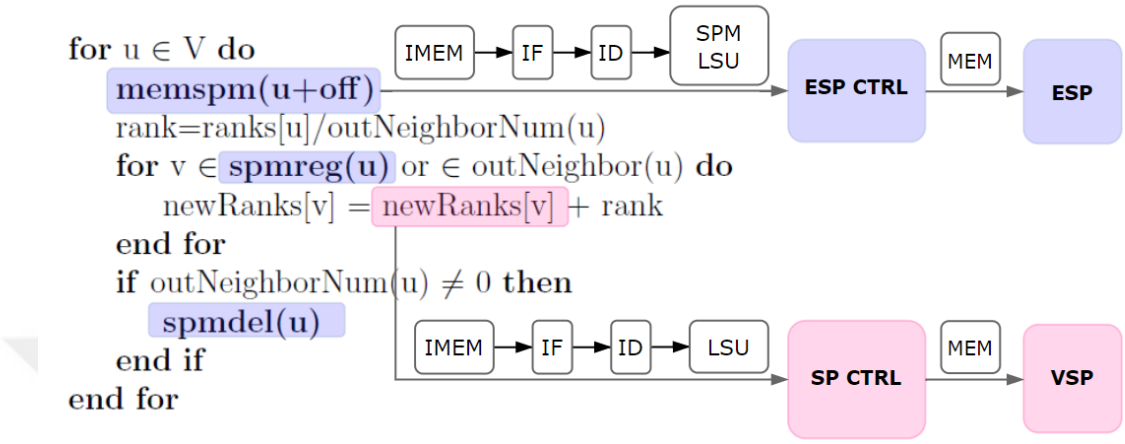

Figure 5.3: A summary of the stages executed by instructions.

In Figure 5.3 we roughly illustrate the route the instructions take on the hardware. Each instruction, custom or not, is first fetched from IMEM. If what is being executed is an instruction like `memspm` which targets ESP, it is directed to the ESP Controller via SPM Load-Store Unit. On the next cycle, the core fetches the next instruction from IMEM as ESP Controller simultaneously handles the job given by the instructions related to ESP. When the program makes a memory request on the VSP Data, SP Controller guarantees that the data is fetched to or updated on VSP. Both controllers access main memory whenever they receive an instruction that requires them to do so.

Chapter 6

Experiments

6.1 Experimental Setup

We ran tests on Vivado Simulator [49] to evaluate our graph processor architecture. The design has been broken down to successive versions shown in Table 6.1 to gauge the scratch-pad performances separately and to demonstrate the evaluation of the graph processor. **GPProc** is the abbreviation of the general-purpose processor we developed by combining Ibex core with a single-level cache adapted from an earlier design [43]. It does not feature any scratch-pads. We compare all versions of the graph processor against this general-purpose processor. **pre-GraphProc-1** is a primitive processor with ESP and a cache, corresponding to architecture given in Figure 4.1. This version only tests the usefulness of ESP, thus lacks VSP and GSP. Similar to pre-GraphProc-1, **pre-GraphProc-2** is the second version of our architecture where VSP and GSP replace the cache. However, this version does not include ESP as shown in Figure 4.6. The total size of VSP and GSP matches the size of the cache of the GPProc to keep the architectures comparable. In line with the observation illustrated in Figure 5.1, we gave VSP and GSP equal sizes. The final processor architecture is the most advanced version, presented as the **Graph Processor**, given in Figure 4.7. It features all of our scratch-pad memories.

Table 6.1: Processor versions tested in experiments.

Architecture	**Cache**	**ESP/Chunksize/Chunk#**	**VSP/GSP**
GPProc	1K	None	None
pre-GraphProc-1	1K	0.5K/32/4	None
pre-GraphProc-2	None	None	0.5K/0.5K
Graph Processor	None	0.5K/32/4	0.5K/0.5K

As benchmarks, we implemented PR, SSSP, and BFS in C language in the way they were given in Chapter 5. We generated three graphs, dubbed G1, G2, and G3, using Graph500 [50] synthetic graph generator. Because of long simulation times, we kept both the cache and the graph sizes small but for our experiments to mimic realistic set-ups we ensured that the ratio between the number of vertices in the graphs and the capacity of the last-level cache in terms of words remained similar to the literature [22, 28]. For all three graphs, the total graph sizes were much bigger than the cache or scratch-pad capacity in all configurations. Table 6.2 lists the features of the graphs we used. All were stored in the CSR [40] format.

Table 6.2: Graphs used for experiments.

Graph	**Vertex#**	**Edge#**	**Degree**	**Directed**
G1	1024	17641	32	Y
G2	2048	39238	32	Y
G3	4096	48367	16	Y

We had the controllers counted the cache and SPM miss numbers because a cache, VSP or GSP miss definitely leads to a datapath-blocking memory access. In architectures where there is no cache, we refer to blocking memory accesses as scratch-pad or **SP** misses as all such accesses necessarily stem from GSP or VSP misses. Blocking memory accesses are abbreviated as **BMA** in the figures for the clarity of the presentation. Whenever relevant, we collected other information such as miss rates or percentage of requests directed at each SPM. Our reports start with the results on pre-GraphProc-1 and pre-GraphProc-2, and continue with the statistics as well as a sensitivity analysis on the Graph Processor.

6.2 Experimental Results

6.2.1 ESP on pre-GraphProc-1

We begin the section with an attempt to understand how much we gain by integrating ESP into the design. The architecture being considered can be observed in Figure 4.1. By preloading some of the edge data to ESP in ways which are algorithmically described in Chapter 5, we hope to decrease the edge traffic flowing through the cache. This should, in turn, reduce the cache misses, or the number of datapath-blocking memory accesses. We compare the cache misses occurring at pre-GraphProc-1 against those at the GPProc and express the percentage decrease in BMA with three graphs for all the benchmarks. We omit the statistics on cache miss rates: They become non-relevant as ESP reduces not only the misses but also the requests linked with the cache.

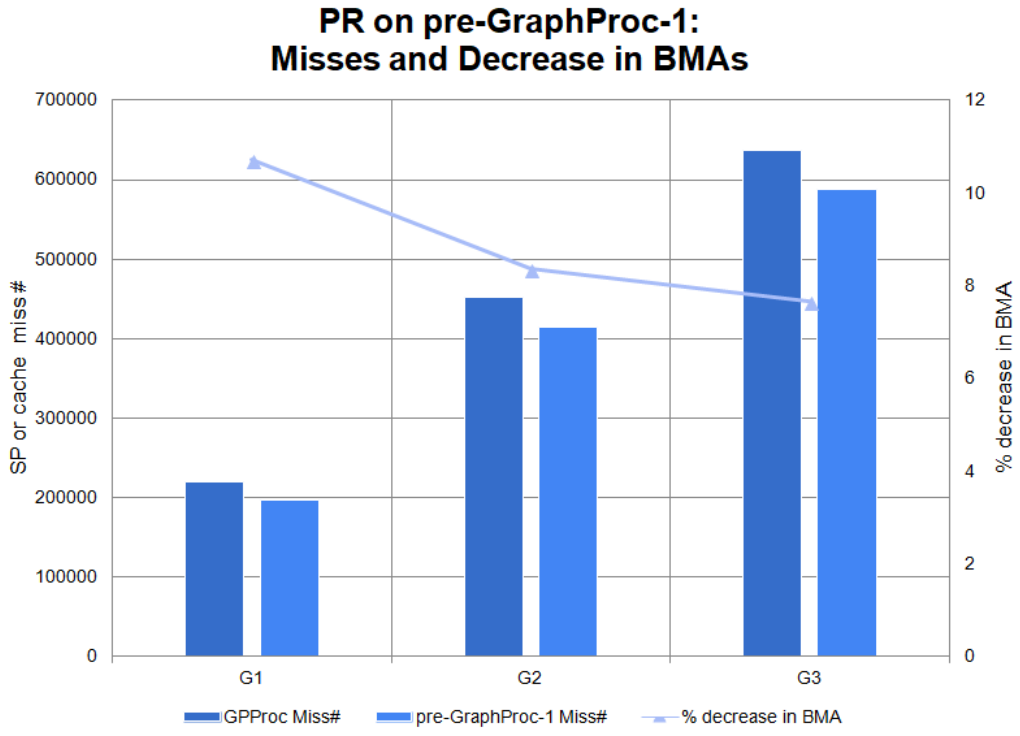

Figure 6.1: PR miss numbers on GPProc and pre-GraphProc-1 caches with percentage decrease in the number of BMAs.

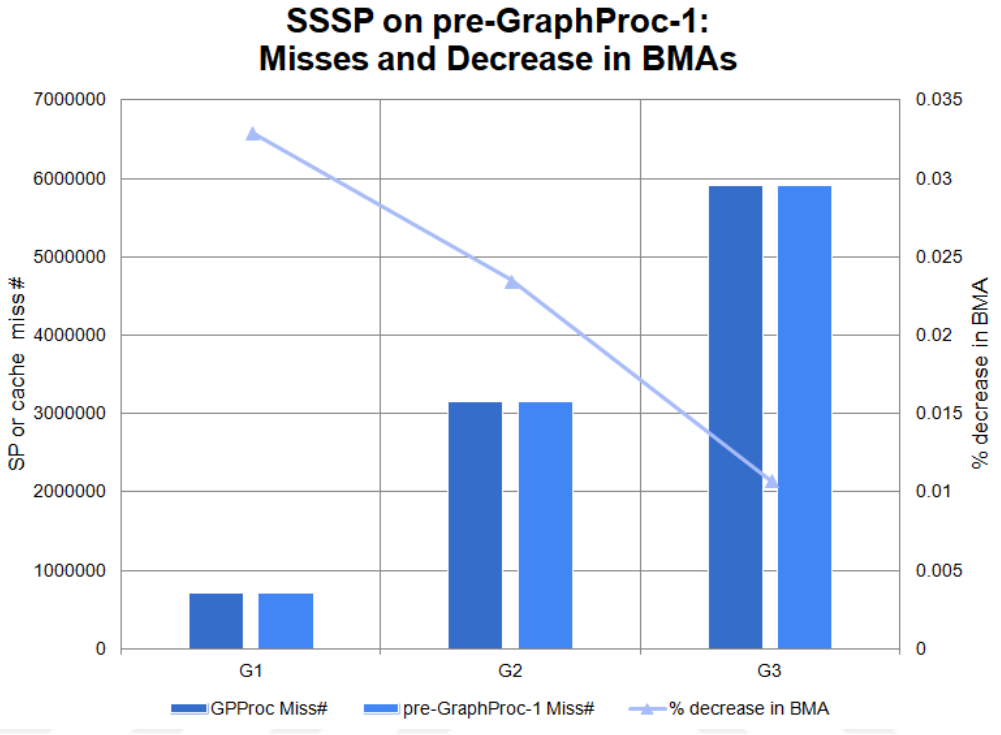

Figure 6.2: SSSP miss numbers on GPProc and pre-GraphProc-1 caches with percentage decrease in the number of BMAs.

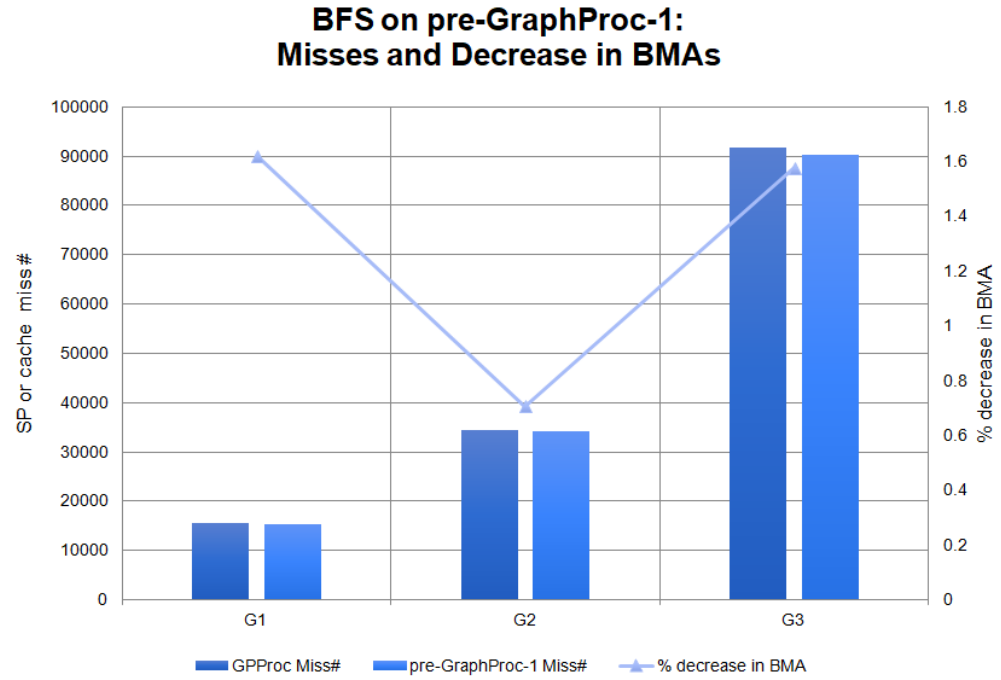

Figure 6.3: BFS miss numbers on GPProc and pre-GraphProc-1 caches with percentage decrease in the number of BMAs.

As can be seen in Figure 6.1, PR exhibits the largest gain around 8% both because of its highly predictable nature and its algorithmic dependence on the edge traffic. The percentage drop in BMAs for SSSP, displayed in Figure 6.2, is negligible below 1% for all graphs because in this algorithm the vertex-related accesses make up for the vast majority of the memory load, leaving little room for improvement on the side of the edge traffic. Figure 6.3 reveals that BFS suffers from the capacity bottleneck as expected, and its utilization of ESP accounts for at most 1.6% decrease in the total number of memory accesses. The design is open for improvement which is attempted with VSP and GSP in the next version.

6.2.2 VSP and GSP on pre-GrapProc-2

In pre-GraphProc-2 we replace the cache with VSP and GSP and omit ESP as in Figure 4.6. Similar to a cache miss of GPProc, each pre-GraphProc-2 request missed at VSP or GSP (collectively referred to as SP miss) results in a blocking memory access, and any decrease in misses implies fewer BMAs. This version mainly helps to evaluate the performance of VSP.

Our results show that VSP and GSP significantly reduce the number of blocking memory accesses in all three of the applications. In PR and SSSP, percentage decrease in BMA generally increases with the graph size: For G1, G2, and G3, Figure 6.4 shows that PR starts with an improvement around 9% and that goes up to 16%, while the drops in SSSP are revealed to be 68.7%, 68.2%, and 72.6% in Figure 6.5. BFS on the other hand, gives 47.7%, 43.5%, and 32.2% which can be seen in Figure 6.6.

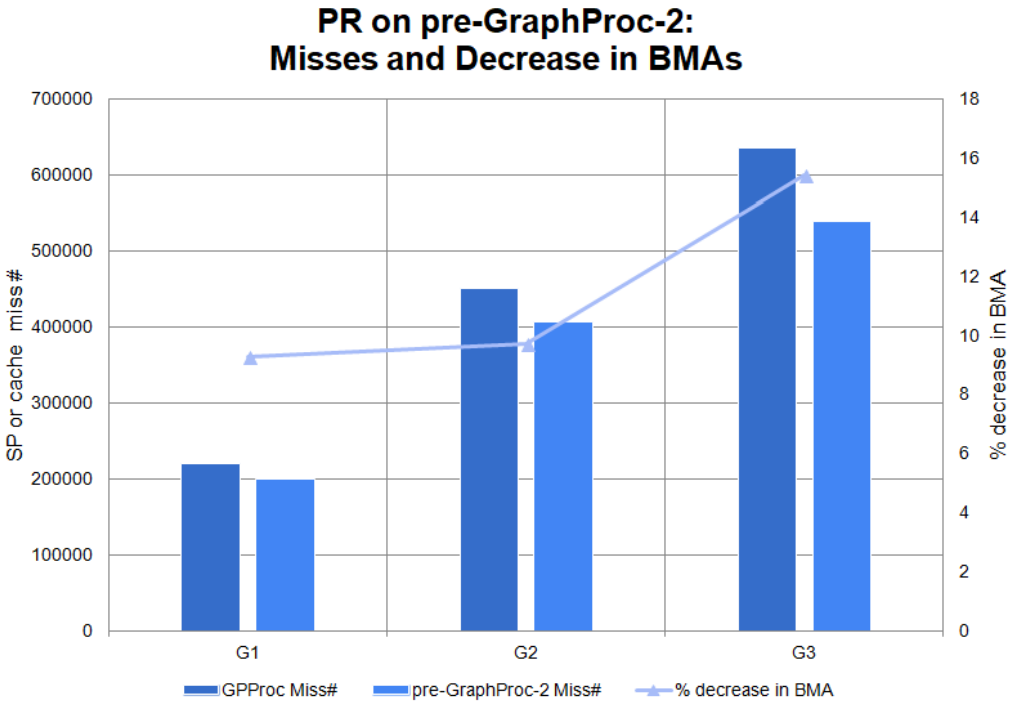

Figure 6.4: PR miss numbers on GPProc cache and pre-GraphProc-2 SP with percentage decrease in the number of BMAs.

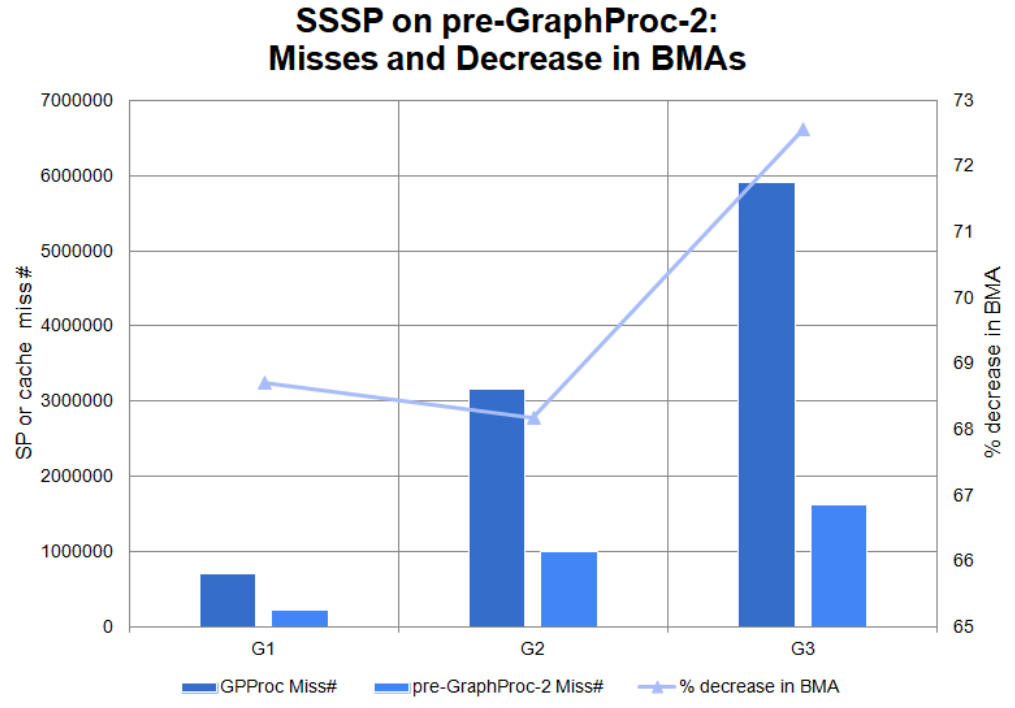

Figure 6.5: SSSP miss numbers on GPProc cache and pre-GraphProc-2 SP with percentage decrease in the number of BMAs.

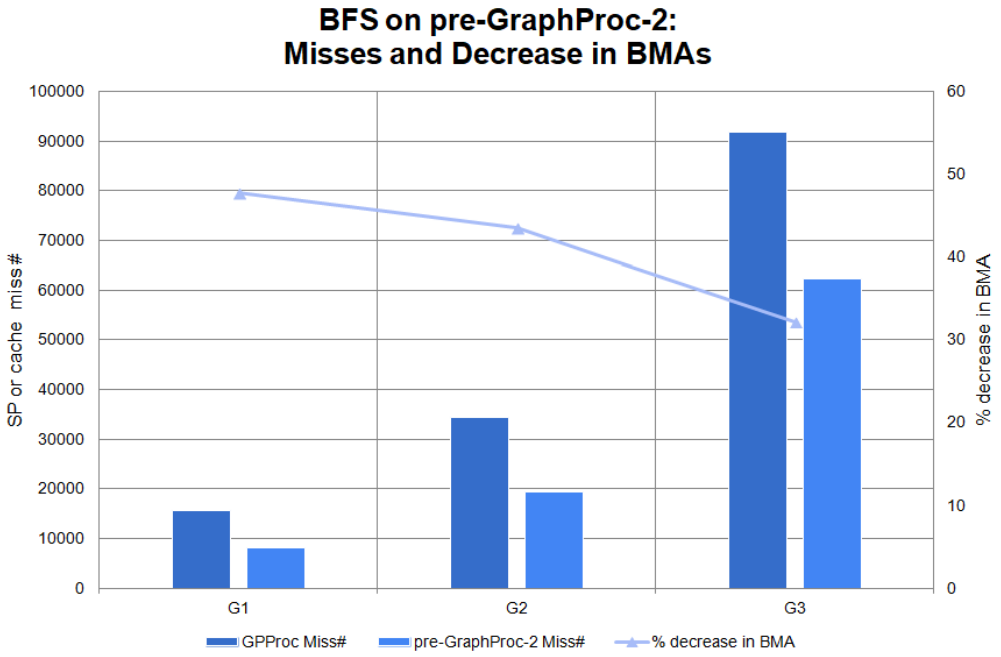

Figure 6.6: BFS miss numbers on GPProc cache and pre-GraphProc-2 SP with percentage decrease in the number of BMAs.

In SSSP, the VSP Data, or the array *set*, is overwhelmingly more frequently accessed than any other range in the memory, which is a purely algorithmic feature of Dijkstra's SSSP that causes the application to enjoy the highest gain. Unlike PR, both BFS and SSSP executions depend on a starting node, which might be resulting in arbitrary patterns of execution that do not necessarily scale with the graph size. The reason why BFS gains less with increasing graph size might again be algorithm specific, as many factors including the degree of the graph and the position of the starting node directly predict some parameters that govern the course of the run such as the number of iterations and vertices activated.

6.2.3 The Graph Processor

This section presents the results of our advanced architecture whose diagram is given in Figure 4.7. The Graph Processor has no cache, but includes ESP, VSP, and GSP as well as all the improvements and additions to the general-purpose

processor's datapath, implemented to support the new instructions packaged under our ISA-level extension. We examine the data relating to the number of blocking memory accesses in this section, while an analysis of how sensitive the Graph Processor is to the *chunksize* and the number of chunks in ESP can be found in the next.

Similar to the previous sections, we first compare the cache misses at the general purpose processor against SP (VSP and GSP) misses of the Graph Processor. We express the percentage decrease in blocking memory accesses for all the benchmarks with three graphs. We also report the cache miss rates in GPProc, SP miss rates in the Graph Processor, the percentages of ESP requests in total memory requests, and ESP miss rates. In portions depending on the application, ESP takes over some of the edge traffic through non-blocking loads, hopefully reducing compulsory and conflict misses which would otherwise occur in GSP. VSP maps to its own exclusive range, leaving GSP more available for other sets of data with higher locality and thus decreasing conflicts.

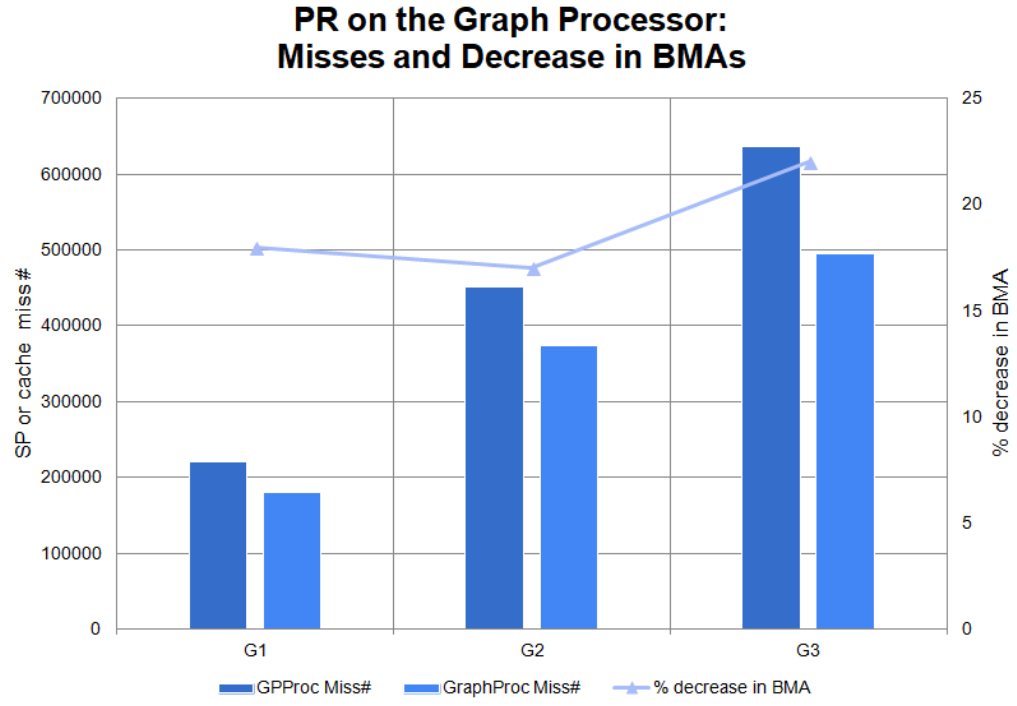

Figure 6.7: PR miss numbers on GPProc cache and the Graph Processor SP with percentage decrease in the number of BMAs.

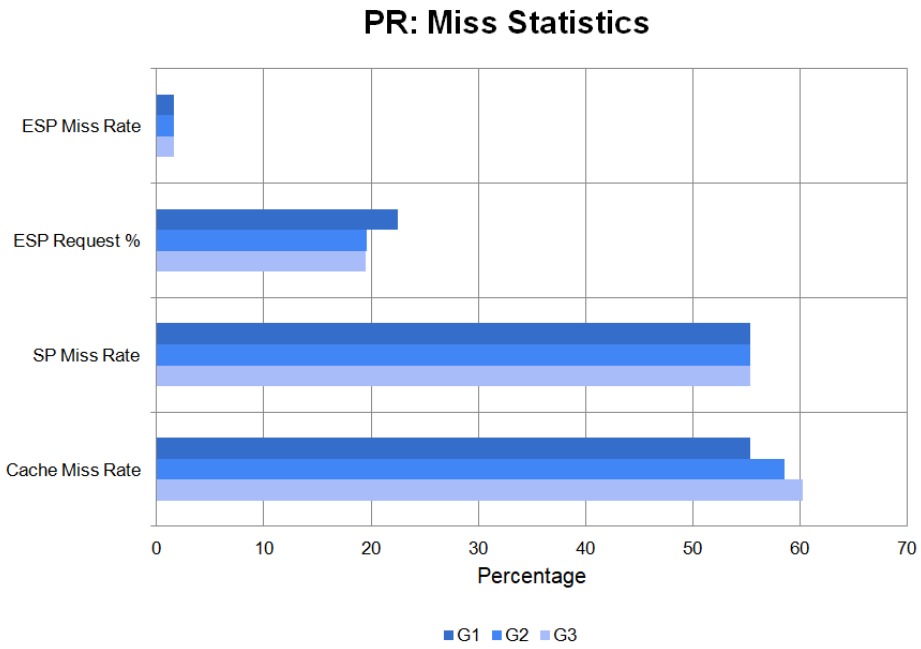

Figure 6.8: Cache, SP, and ESP miss rates with ESP Utilization for PR.

For PR, Figure 6.7 shows 18%, 17%, and 22.4% decrease in blocking memory accesses for G1, G2, and G3, respectively. This seems to be the cumulative gain from independent ESP and VSP structures. PR also utilizes ESP nicely, directing almost 20% of its requests at ESP which enjoys high hit rates as demonstrated in Figure 6.8. Such an ESP utilization should correspond to an important decrease in requests made to GSP. Despite the fact that number of requests targeting GSP is lower, the number of misses drops so much that we are able to observe a general decrease when we look at GPProc cache and the Graph Processor SP miss rates.

Results regarding SSSP are given in Figures 6.9 and 6.10. Figure 6.9 displays a significant improvement around 70% for this application, and pretty much all of it stems from VSP as can be inferred looking at Figure 6.5. Since miss numbers decrease about 70% and the ESP utilization is low, all of this gain in miss numbers is reflected on the miss rates which also drop from 90% to 30% level, as shown in Figure 6.10. ESP again enjoys high hit rates, but for SSSP the edge traffic makes up a portion of memory accesses that is so tiny that the benefits of ESP remain negligible.

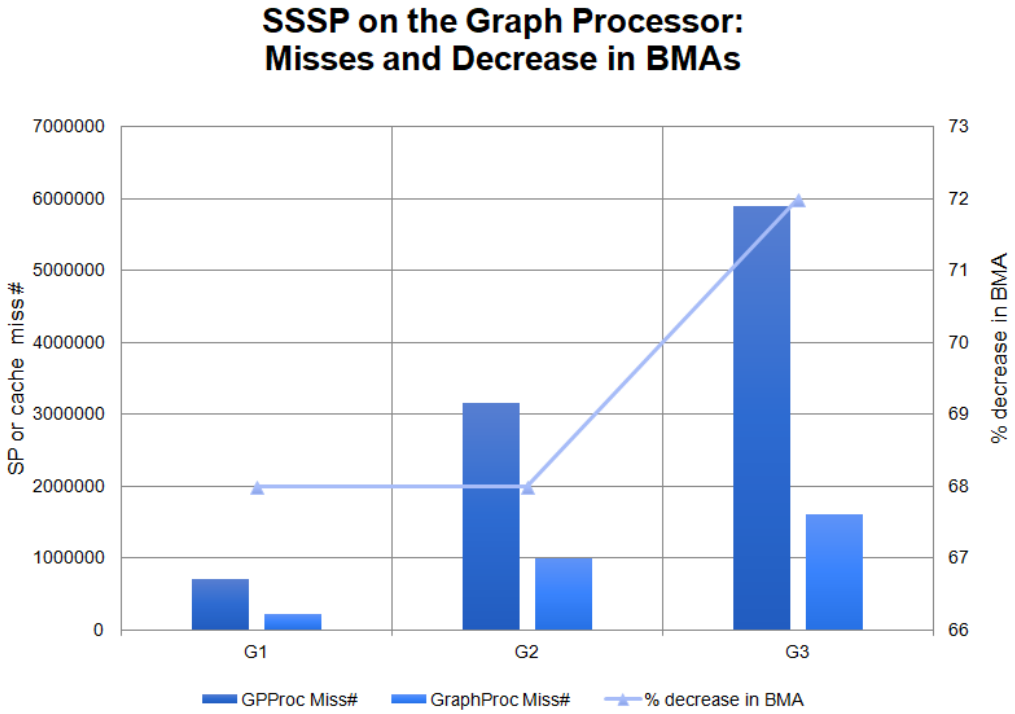

Figure 6.9: SSSP miss numbers on GPProc cache and the Graph Processor SP with percentage decrease in the number of BMAs.

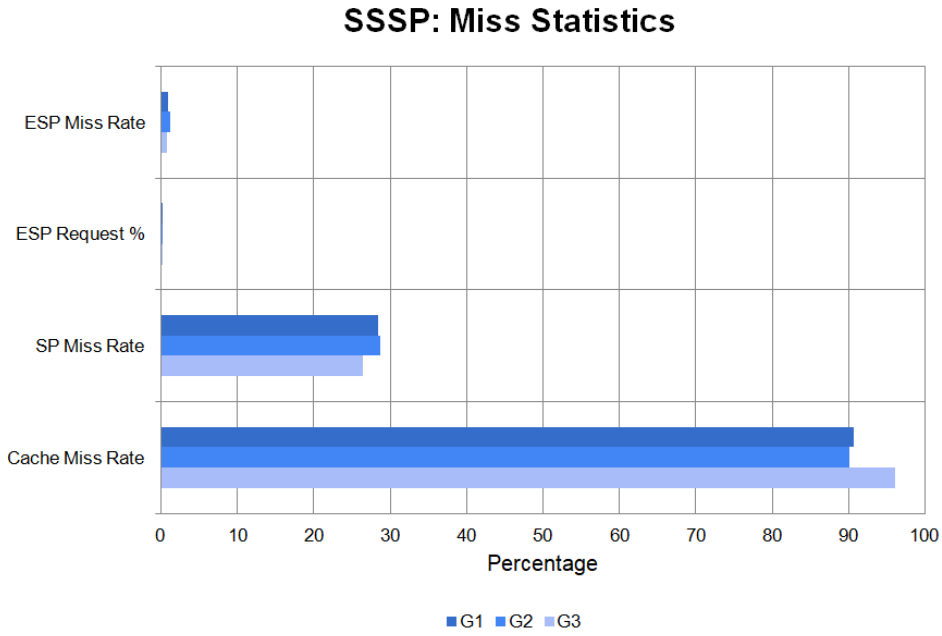

Figure 6.10: Cache, SP, and ESP miss rates with ESP Utilization for SSSP.

The last application we test is BFS which makes on average 40% fewer BMAs on the Graph Processor as shown in Figure 6.11. However, BFS does not suit well for ESP as its active list used for predicting which vertices will be needed soon ramps up too rapidly for ESP to accommodate. Figure 6.12 displays that ESP miss rates for BFS are indeed unusually high. When the initial request for any vertex made at ESP is missed, the software infers that ESP does not contain the data associated with the vertex and avoids making further requests on it, reducing the ESP utilization. Further, the capacity bottleneck becomes tighter as the graph size increases, which is the main reason why the decrease in blocking memory accesses does not scale well with growing graph sizes. Still, migrating from the cache to scratch-pads result in an observable improvement in hit rates, mainly because the application overall experiences less conflicts thanks to VSP.

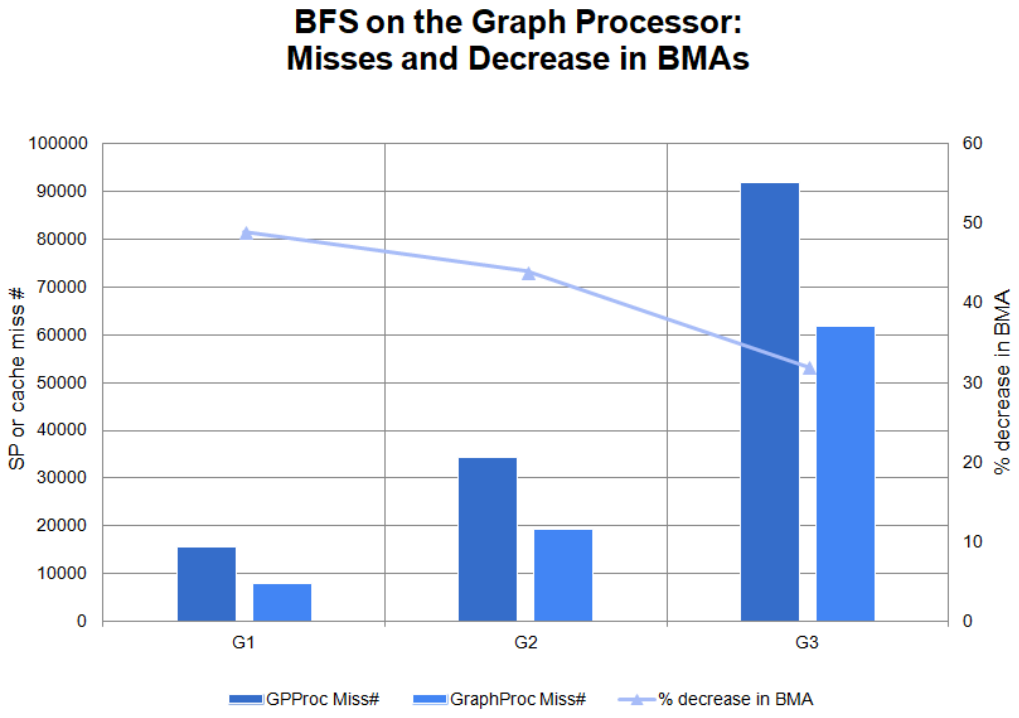

Figure 6.11: BFS miss numbers on GPProc cache and the Graph Processor SP with percentage decrease in the number of BMAs.

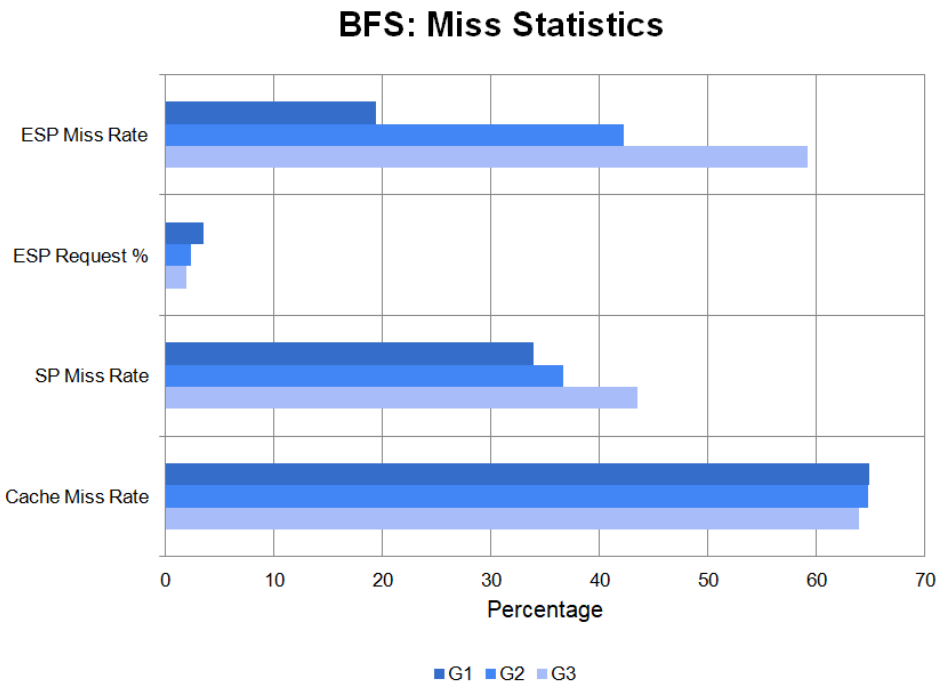

Figure 6.12: Cache, SP, and ESP miss rates with ESP Utilization for BFS.

6.3 Sensitivity Analysis

In this section, we briefly analyze how the number of blocking memory accesses changes with the *chunksize* and the number of chunks in the ESP.

6.3.1 ESP Chunksize

In general, ESP hit rates should increase as the *chunksize*, dubbed as *cs*, increases because larger chunks enable the software to preload more neighbors with each `memspm` instruction. The original size of the ESP chunks in the Graph Processor is 32 words. Keeping the *chunksize* as a power of two, we test the design for cs = 16 and 64 as well. We give the number of SP misses and the percentage decrease in the number of blocking memory accesses relative to GPProc on separate figures.

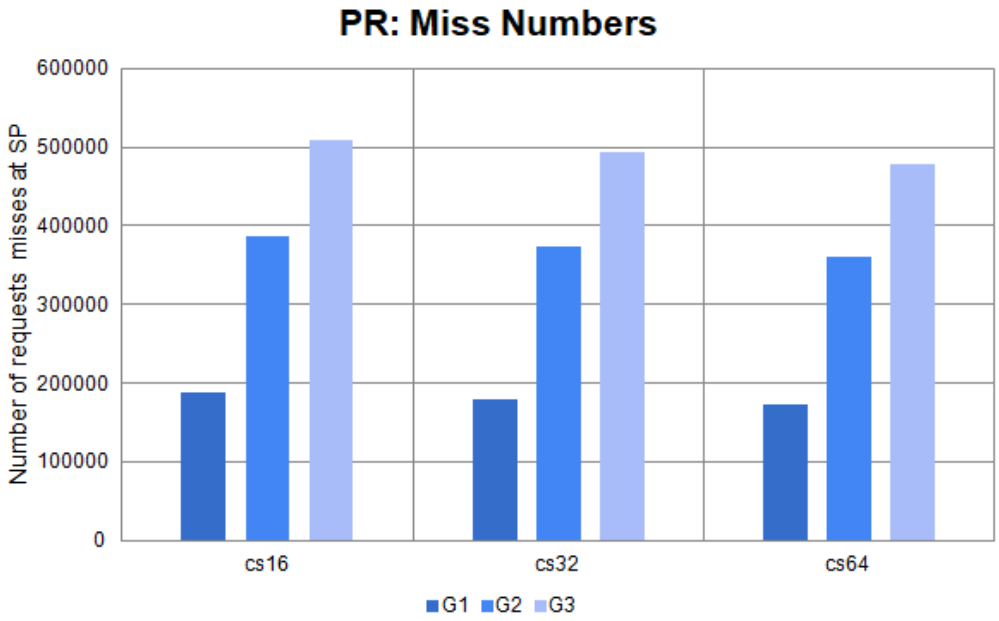

Figure 6.13: Miss numbers of PR on the Graph Processor with cs = 16,32,64.

For PR, as cs increases, the number of misses drops in observable amounts as can be seen in Figure 6.13. This is directly reflected in our target parameter, the number of BMAs, as the percentage of decrease in BMAs relative to GPProc increases with increasing cs. The general trend can be observed in Figure 6.14. As the *chunksize* grows from 16 to 64; G1, G2, and G3 respectively make 15% to 21%, 14% to 20%, and 20% to 25% fewer BMAs. PR turns out to be the application that makes the most out of larger *chunksize*s because it uses ESP more than others do. Hence, the software is able to carry out an important amount of its edge traffic (around 20% of the application's entire memory traffic as can be seen in Figure 6.8) through non-blocking loads, and the more words fetched with each chunk the better for reducing the blocking memory accesses.

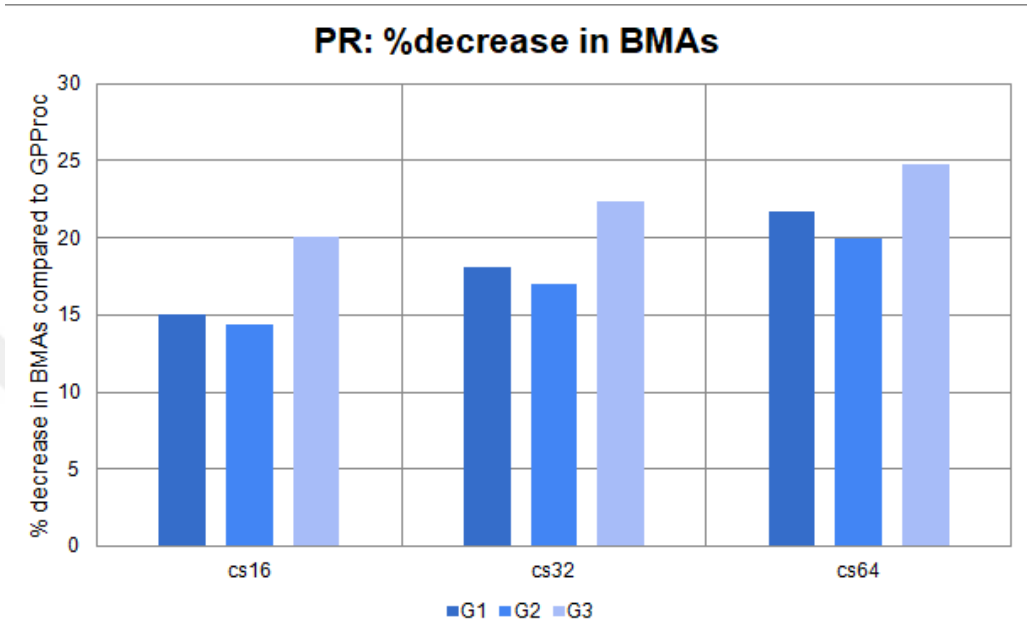

Figure 6.14: Percentage decrease (compared to GPProc) in BMA of PR on the Graph Processor with cs = 16,32,64.

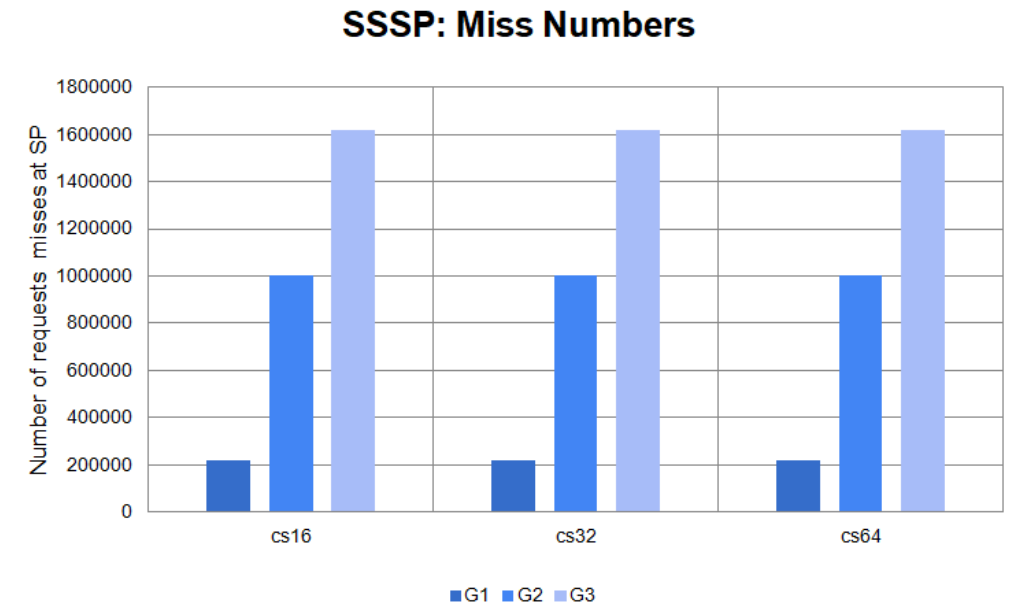

Figure 6.15: Miss Numbers of SSSP on the Graph Processor with cs=16,32,64.

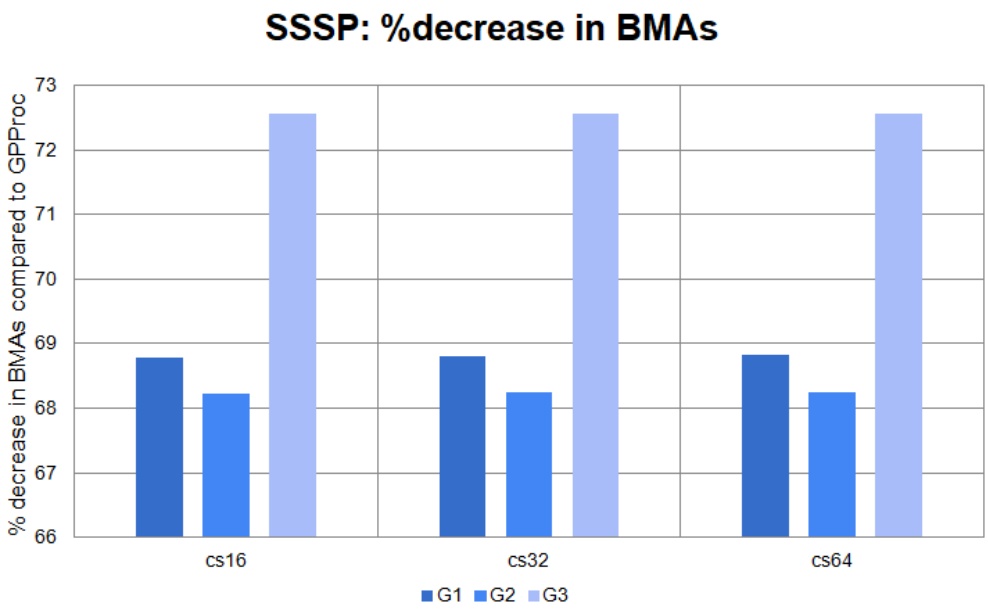

Figure 6.16: Percentage decrease (compared to GPProc) in BMA of SSSP on the Graph Processor with cs=16,32,64.

In SSSP, both the number of misses and the relative BMA drop percentages do not change significantly as shown in Figures 6.15 and 6.16. For G1, the decrease in BMAs goes from 68.7% to 68.8%, while for G2 and G3 the rates stay around 68.2% and 72.5%. This once more establishes that in applications where the edge traffic makes up a relatively tiny portion of the memory accesses, VSP performance becomes much more influential and even when the predictability is high and ESP miss rates are thus low (referring to Figure 6.9 on the statistics related to SSSP), increasing the number of edges preloaded to ESP is not worthwhile.

Our final benchmark is BFS which, despite its moderate dependence on the edge traffic (reflected in high percentage of GSP requests in Figure 5.1), is unable to utilize ESP well due to the software being unable to order the preloads in a fashion more considerate of ESP. The miss numbers given in Figure 6.17 stay pretty much constant while BMAs decrease about 1% for G1 and 0.5% for G2 and G3 as illustrated in Figure 6.18.

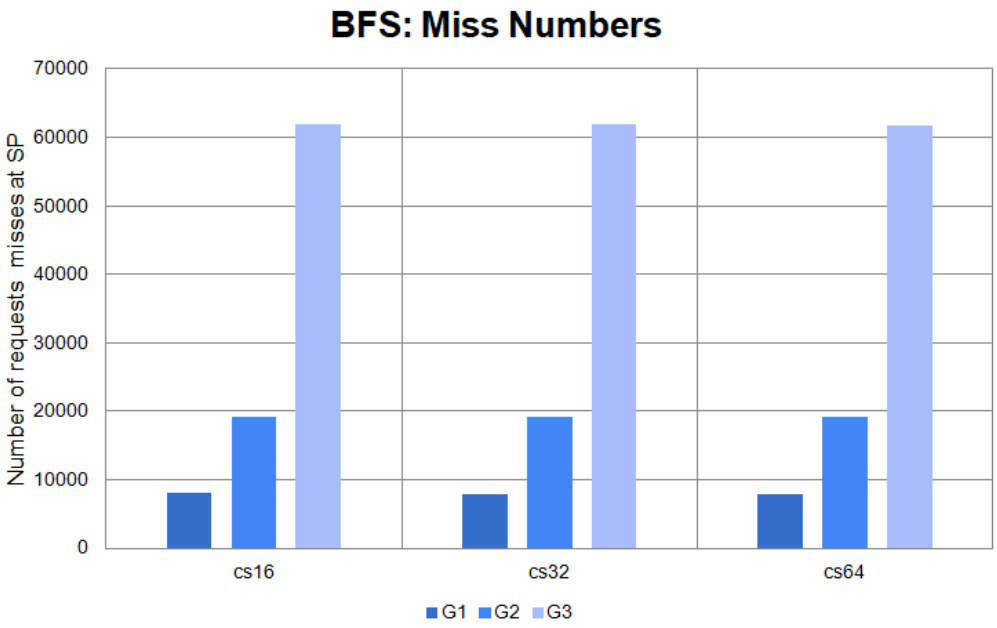

Figure 6.17: Miss Numbers of BFS on the Graph Processor with cs=16,32,64.

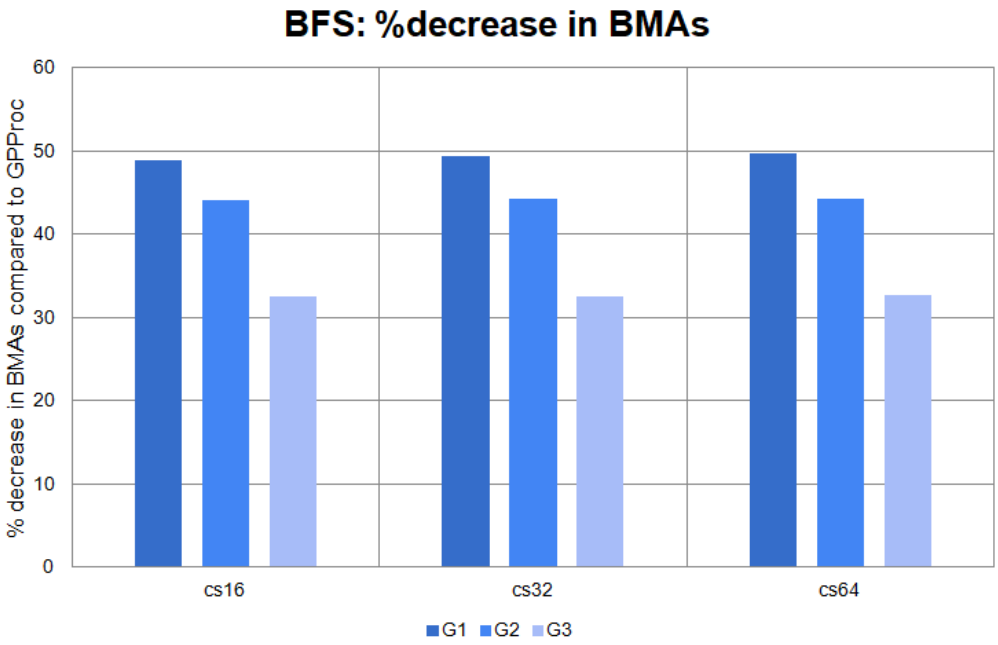

Figure 6.18: Percentage decrease (compared to GPProc) in BMA of BFS on the Graph Processor with cs=16,32,64.

6.3.2 ESP Chunk Number

The main purpose of this analysis is to verify that ESP capacity by itself is not a bottleneck, and while VSP and GSP (similar to a general-purpose cache) should get bigger for larger data, ESP size can remain small. This is because ESP, manipulated by the software, is read out in the order it is filled in and does not need to accommodate unnecessary data. This property caused PR and SSSP to give the same miss and BMA numbers when ESP capacity was altered between 4, 8, and 16 chunks.

On the other hand, BFS has slightly improved as shown in Figures 6.19 and 6.20 for it was already suffering from misses occurring at the first read attempt related to each chunk. Therefore, as the ESP capacity grow to include more distinct chunks, some of such misses were eliminated. However, the main reason why BFS performs poorly is the capacity of the new arrival queue, a small FIFO buffer independent from the ESP size. This bottleneck kept the reduction of BMAs small, around 0.5% for all three graphs. Therefore, it is best to keep the number of chunks minimum at 4, as it is for the Graph Processor. We can conclude that the number of chunks in ESP is not a significant parameter for the performance of the architecture, and the ESP size can remain small when the system scales up for possible future applications.

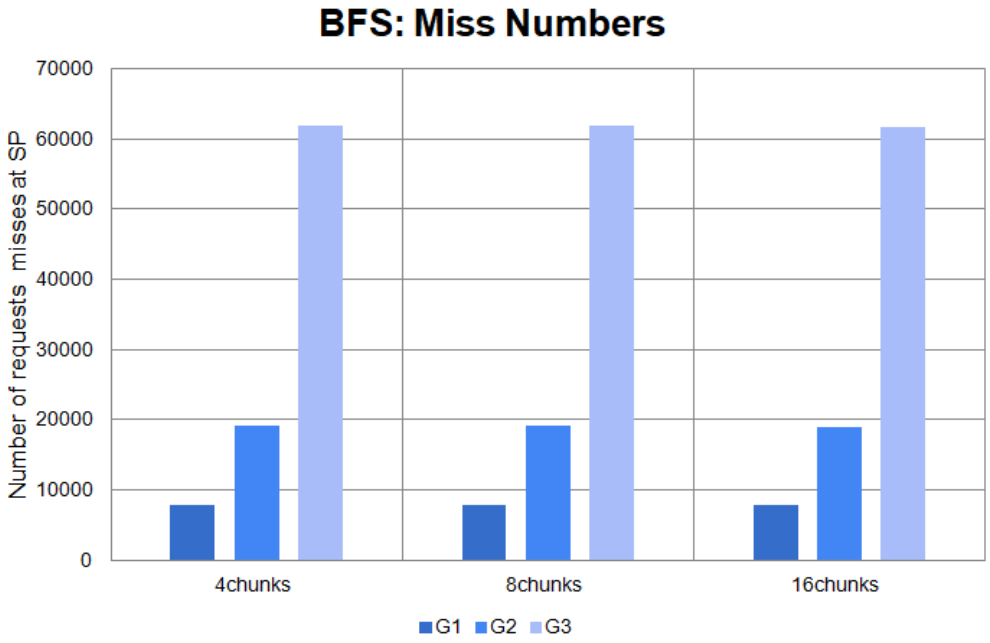

Figure 6.19: Miss Numbers of BFS on the Graph Processor with ESP number of chunks 4,8,16.

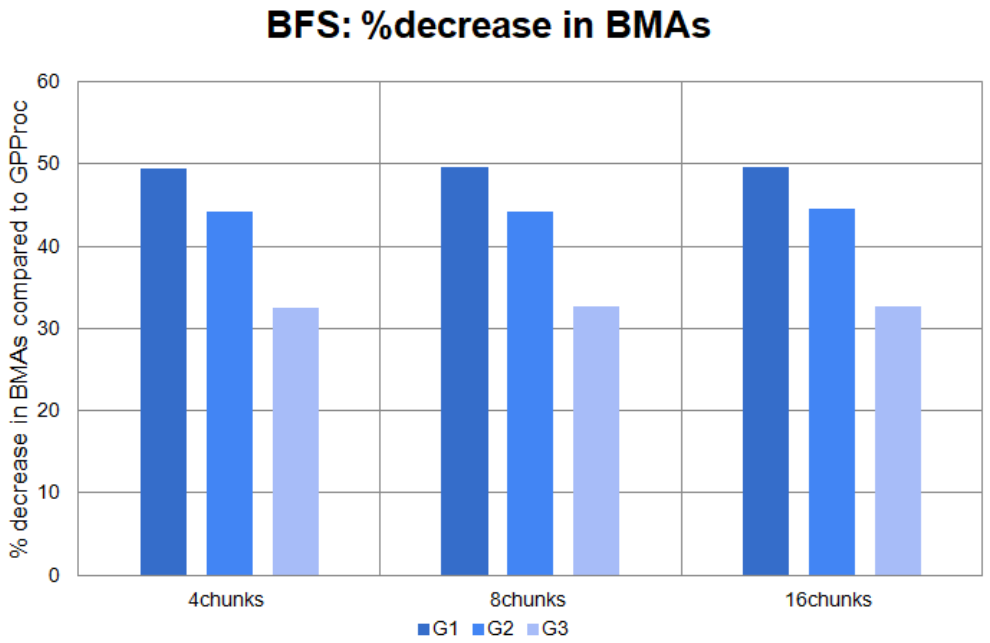

Figure 6.20: Percentage decrease (compared to GPProc) in BMA of BFS on the Graph Processor with ESP number of chunks 4,8,16.

Chapter 7

Discussion

This chapter includes a summary of our findings, a critique of our architecture, and a discussion about possible future work.

Our experiment results indicate that the proposed graph processor decreases the number of datapath-blocking memory accesses (BMAs) for all the benchmarks we considered. Reducing the number of blocking memory accesses is important because they cause the datapath to be stalled and they therefore directly predict the runtime or the performance of the application. For PageRank, Single-Source Shortest Path, and Breadth-First Search, the decrease in BMAs are about 20%, 70%, and 40%, respectively.

Nevertheless, some limitations of our architecture keep it from being consistently helpful in all graph applications. First, despite being the most complicated scratch-pad, ESP will fail to make meaningful contributions to the performance of BFS-like applications where the predictor used for loading the scratch-pad is challenging for its architecture. The size of the new arrival queue used to temporarily accommodate the addresses of the edges to be loaded is not enough for the software. Either the way the software is written in such applications should be modified, or another mechanism allowing `memspm` to function better must be built. Second, the architecture might have been improved with another strictly

software-controlled scratch-pad for vertex traffic. VSP is successful in decreasing the conflicts and the blocking memory accesses caused by a certain portion of vertex data, but it is loaded with usual load-word instructions and hence it cannot avoid compulsory misses. Designing a structure like ESP for the vertex traffic would require complexity because of the poor locality of the vertex data and the coherency issues stemming from its read/write nature, but the structure might have turned out to be useful enough to be worth such burdens. One possible future work might deal with advancing the architecture in these two aspects.

Another work is to scale the architecture proposed here up for more sophisticated systems. While the graph processor design we presented is well-suited for embedded systems with size and energy constraints, it can easily be adapted for large-scale systems because its hardware components are portable and the software front is simple to implement. In such larger systems, we recommend the entire cache hierarchy (spanning L1 to L3 caches) to be replaced by VSP and GSP at each level. The relative sizes of the two might be varied depending on further analysis of the graph applications or on the characteristics of the particular system architecture being considered. For ESP, a small portion of the L1 space must be allocated. ESP size can remain tiny compared to the entire hierarchy composed of VSP and GSP because it is not required to be large to contribute to the performance of the architecture. As for the core, any RISC-V processor should be adequate, and only a few trivial modifications on the decode and execute stages are needed to support our ISA-level extensions. The SPM Load-Store Unit might be implemented inside the core as we have done or as a separate module serving in between the SPM controllers and the datapath. The software should remain the same across different systems, which arises as an important advantage providing ease of programmability. Any agent with our modified RISC-V C compiler can produce the binaries utilizing the graph processor.

Other possible future works include implementing the design on different FPGA platforms and optimizing the software. An FPGA implementation should be useful in gauging the performance of the architecture with further analysis on runtime and power. It is also a necessary step before offering the design in ready-to-use chips. As for the software, its intelligence directly affects the gain

from ESP and VSP, so any further work on optimizing the way the high-level code is written should be worth the time and effort. Specifically, the use of `memspm` instruction might change to better fit the nature of the applications. VSP might be allocated to alternating portions in the memory throughout the runs. Once set and extended, the particular rules of writing each graph application for the graph processor might be formally documented and released to guide the users.

Chapter 8

Conclusion

In this thesis, we present a novel domain-specific RISC-V processor for graph applications. The processor includes specialized SPMs and it supports a custom ISA-level extension. The main purpose of the architecture is to increase the performance of graph algorithms by executing them with fewer datapath-blocking memory accesses than general-purpose processors do.

First, we considered the common features of iterative and vertex-centric graph algorithms to design custom SPMs fitting their needs. Specifically, we targeted the memory bottleneck, and we made a distinction between the edge and vertex traffic with ESP and VSP. Complementing the two with GSP, we replaced underperforming caches with our SPMs. Next, we reviewed the RISC-V ecosystem, found a core that can be modified, and extended its hardware to include the new functionality we designed.

For the architecture to be controlled by the software, we extended RISC-V ISA with custom instructions. We reflected the architectural extension on the datapath of our graph processor and on the RISC-V compiler. With the compiler support provided, we modified well-known graph algorithms, PageRank, Single-Source Shortest Path, and Breadth-First Search, both to demonstrate how the graph processor should be utilized by high-level code and to test our framework.

Finally, we compared our architecture against a general-purpose processor created by combining our base core with a conventional cache. With the graph processor, the number of blocking memory accesses made by PageRank, Single-Source Shortest Path, and Breadth-First Search algorithms on average decreased by 20%, 70%, and 40%, respectively.

In conclusion, the graph processor architecture we designed can reduce the number of datapath-blocking memory accesses in graph algorithms. The software should take advantage of our custom instructions and it should abide by the demonstrated rules of using ESP and VSP. We expect our design to be a useful contribution to the efforts in increasing the performance of current and future graph analytics applications.

Bibliography

[1] G. Pavlopoulos, M. Secrier, C. Moschopoulos, T. Soldatos, S. Kossida, J. Aerts, R. Schneider, and P. Bagos, "Using graph theory to analyze biological networks," *BioData mining*, vol. 4, pp. 1–27, 2011.

[2] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 78–88, 2011.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: bringing order to the web," Technical Report 1999-66, Stanford InfoLab, November 1999.

[4] U. Brandes, "A faster algorithm for betweenness centrality," *The Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

[5] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, pp. 30–37, Aug 2009.

[6] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, 2014.

[7] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[8] W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, "Novel graph processor architecture, prototype system, and results," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2016.

[9] A. Lumsdaine, D. P. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Process. Lett.*, vol. 17, pp. 5–20, 2007.

[10] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Commun. ACM*, vol. 59, pp. 78–87, 2016.

[11] N. Kapre, "Custom FPGA-based soft-processors for sparse graph acceleration," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 9–16, 2015.

[12] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *2015 IEEE International Symposium on Workload Characterization*, pp. 56–65, 2015.

[13] G. Cong and K. Makarychev, "Optimizing large-scale graph analysis on multithreaded, multicore platforms," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 414–425, 2012.

[14] "RISC-V." https://github.com/riscv. Accessed: 2020-07-01.

[15] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, 2016.

[16] S. Yesil, M. M. Ozdal, T. Kim, A. Ayupov, S. Burns, and O. Ozturk, "Hardware accelerator design for data centers," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 770–775, 2015.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pp. 135–146, ACM, 2010.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010.

[19] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," *Proc. VLDB Endow.*, vol. 8, p. 281–292, Nov. 2014.

[20] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a pc," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, (Hollywood, CA), pp. 31–46, USENIX Association, Oct. 2012.

[21] S. Zhou and V. K. Prasanna, "Accelerating graph analytics on CPU-FPGA heterogeneous platform," in *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 137–144, 2017.

[22] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pp. 105–117, ACM, 2015.

[23] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, S. Burns, and O. Ozturk, "Architectural requirements for energy efficient execution of graph analytics applications," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 676–681, 2015.

[24] S. Glinski, T. Lalumia, D. Cassiday, T. Koh, C. Gerveshi, G. Wilson, and J. Kumar, "A processor for graph search algorithms," in *1987 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 162–163, 1987.

[25] M. Scheevel, "NORMA: A graph reduction processor," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, (New York, NY, USA), p. 212–219, Association for Computing Machinery, 1986.

[26] J. Zhang, S. Khoram, and J. Li, "Boosting the performance of FPGA-based graph processor using hybrid memory cube: A case for breadth first search," in *FPGA '17*, pp. 207–216, 02 2017.

[27] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "GraphGen: An FPGA framework for vertex-centric graph computation," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 25–28, 2014.

[28] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 140–149, 2014.

[29] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The RISC-V instruction set manual," 2014.

[30] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," Tech. Rep. UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

[31] "Available hardware." https://riscv.org/risc-v-cores/. Accessed: 2020-07-01.

[32] "RISC-V software ecosystem overview." https://riscv.org/software-status/. Accessed: 2020-07-01.

[33] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-T: A RISC-V processor with light weight security extensions," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, HASP '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[34] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal RISC extensions for isolated execution," *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 564, 2015.

[35] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for internet-of-things applications," in *2017*

27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS).

[36] "Ibex RISC-V core." https://github.com/lowRISC/ibex. Accessed: 2020-08-01.

[37] "Ibex user manual." https://ibex-core.readthedocs.io/. Accessed: 2020-06-20.

[38] Lian Li, Lin Gao, and Jingling Xue, "Memory coloring: a compiler approach for scratchpad memory management," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pp. 329–338, 2005.

[39] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, pp. 73–78, 2002.

[40] R. W. Vuduc, *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, 2003. AAI3121741.

[41] S. Beamer, *Understanding and Improving Graph Algorithm Performance*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2016.

[42] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.

[43] D. A. Patterson and J. L. Hennessy, "Computer organization and design, fifth edition: The hardware/software interface," 2013.

[44] "RISC-V GNU compiler toolchain." https://github.com/riscv/riscv-gnu-toolchain/. Accessed: 2020-08-10.

[45] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pp. 166–177, IEEE Press, 2016.

[46] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[47] J. Park, M. Penner, and V. K. Prasanna, "Optimizing graph algorithms for improved cache performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 9, pp. 769–782, 2004.

[48] E. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik*, no. 1, pp. 269–271, 1959.

[49] T. Feist, "Vivado design suite," 2012.

[50] D. A. Bader, J. Berry, S. Kahan, R. Murphy, E. J. Riedy, J. Willcock, A. Korzh, and M. Zalewski, "Graph500." https://graph500.org/.