# EDAN: Towards Understanding Memory-Level Parallelism and Latency Sensitivity of Real-World Applications

Siyuan Shen
ETH Zürich
Switzerland
siyuan.shen@inf.ethz.ch

Mikhail Khalilov
ETH Zürich
Switzerland
mikhail.khalilov@inf.ethz.ch

Lukas Gianinazzi
ETH Zürich
Switzerland
lukas.gianinazzi@inf.ethz.ch

Timo Schneider
ETH Zürich
Switzerland
timo.schneider@inf.ethz.ch

Marcin Chrapek
ETH Zürich
Switzerland
marcin.chrapek@inf.ethz.ch

Jai Dayal
Samsung
USA
jai.dayal@samsung.com

Manisha Gajbe
Samsung
USA
m.gajbe@samsung.com

Robert Wisniewski
Samsung
USA
r.wisniewski@samsung.com

Torsten Hoefler
ETH Zürich
Switzerland
torsten.hoefler@inf.ethz.ch

## Abstract

Resource disaggregation is a promising technique for improving the efficiency of large-scale computing systems. However, this comes at the cost of increased memory access latency due to the need to rely on the network fabric to transfer data between remote nodes. As such, it is crucial to ascertain an application's memory latency sensitivity to minimize the overall performance impact. Existing tools for measuring memory latency sensitivity often rely on custom ad-hoc hardware or cycle-accurate simulators, which can be inflexible and time-consuming. To address this, we present *EDAN* (Execution DAG Analyzer), a novel performance analysis tool that leverages an application's runtime instruction trace to generate its corresponding execution DAG. This approach allows us to estimate the latency sensitivity of sequential programs and investigate the impact of different hardware configurations. EDAN not only provides us with the capability of calculating the theoretical bounds for performance metrics, but it also helps us gain insight into the memory-level parallelism inherent to any real-world application. We apply EDAN to applications and benchmarks such as PolyBench, HPCG, and LULESH to unveil the characteristics of their intrinsic memory-level parallelism and latency sensitivity.

## CCS Concepts

• **Computing methodologies → Modeling and simulation**; • **Modeling and simulation → Model developement and analysis**.

## Keywords

Performance modeling, disaggregated memory, latency sensitivity analysis, simulation, parallel programming

## 1 Introduction

Modern communication networks offer increasingly high bandwidths, following an exponential trend witnessed by the doubling of Ethernet switch rates every two years [28]. Yet, those higher bandwidths are achieved by driving higher frequencies on the links as well as more complex signaling (e.g, PAM4). This leads to higher bit error rates in transceivers that are corrected with stronger and more complex forward error correction (FEC) mechanisms. For example, the upcoming 800G and 1.6T IEEE P802.3df specification [22] supports concatenated and segmented FEC mechanisms that significantly increase the processing latency. Today's fast FEC implementations could be as low as 50ns while future complex FECs could easily increase that latency by 100ns, leading to per-link hop traversal latencies of several hundred nanoseconds [28]. Similar observations apply to all modern networks and continue the trend toward higher bandwidths at the cost of higher latencies.

Resource disaggregation is a promising technique that has been recently explored in both the fields of High-Performance Computing (HPC) and datacenter designs. It challenges the traditional monolithic server architecture by separating heterogeneous resources into discrete units connected by a high-speed network. Not only does this approach allow flexible and dynamic provisioning of resources to better match various application requirements, but it also minimizes the idle time of expensive resources, such as accelerators and CPUs, which greatly reduces the cost and energy consumption in a large system [46]. While memory disaggregation networks opt for lower-latency protocols and weaker FEC protection, they follow the same general trend as Ethernet: higher bandwidth will likely cause higher latency. Thus, the ratio of bandwidth to latency will worsen in the coming generations.

Memory disaggregation is prevalent amongst all the resource aggregation systems [17, 41, 42, 52], as it increases the memory utilization across datacenters and helps boost the scalability of memory-intensive applications, such as data-processing frameworks and HPC applications [51]. However, growing latencies may limit or even reduce their efficiency because data access relies on the network fabric [51]. Gao et al. [24] show that additional latency reduces the performance of data-intensive applications regardless of the network bandwidth. In essence, the more sensitive an application is toward memory latency, the more noticeable its performance degradation will be. To this end, it is crucial to ascertain the memory latency sensitivity and tolerance of applications so that resource allocations and system design can be done in a way that minimizes the overall performance impact.

Measuring memory latency sensitivity, however, is a complex topic on its own. It usually involves artificially injecting latency into memory accesses, recording the relevant performance metrics of applications, and extrapolating application runtime under varying degrees of additional memory latency. As exemplified by the works of Patke et al. [51] and Domke et al. [20], one has the option to depend either on some custom ad-hoc hardware that is inflexible and difficult to acquire or cycle-accurate simulators that are extremely time-consuming.

To address these issues, we present *EDAN* (Execution DAG Analyzer), a novel performance analysis tool that takes advantage of the runtime instruction trace of a sequential application to generate its corresponding execution DAG (eDAG), from which numerous performance metrics can be computed and analyzed. This approach reveals an application's latency-hiding potential by exposing the true dependencies between instructions, which, with the help of a CPU and cache model, allows us to estimate the latency sensitivity of an arbitrary application in a fine-grained manner. In addition, unlike other approaches that require parameter sweeps and executing the same application multiple times, EDAN only needs an application to be executed once. As soon as a program's runtime instruction trace is collected, its eDAG can be generated automatically. At this point, we can easily investigate the impact of different hardware configurations (e.g., cache sizes, number of memory issue slots, etc.) by analyzing the eDAG with various parameters.

Under the assumption of an idealized computational model, EDAN not only provides us with the capability of calculating the theoretical bounds for performance metrics such as bandwidth utilization and memory latency sensitivity, but it also helps us gain preliminary insights into the memory-level parallelism inherent to real-world applications. This, in turn, can guide design decisions about architectural parameters, such as the number of issue slots and cache sizes. We apply EDAN to applications and benchmarks such as PolyBench, HPCG, and LULESH to shed some light on the characteristics of their memory-level parallelism and latency sensitivity.

The primary contributions of our paper are as follows:

- We develop EDAN, an experimental tool for theoretical performance analysis based entirely on execution DAGs.
- We define a new memory cost model inspired by Brent's lemma, which defines upper and lower bounds for the memory access cost of an eDAG based on the number of memory issue slots. From this model, we then derive two performance metrics that quantify the memory latency sensitivity of an application.
- We demonstrate the effectiveness of our tool by applying it to several common HPC applications as well as benchmarks, and present the insights gained from this investigation.

## 1.1 Motivation

To assess the memory latency sensitivity of an application, state-of-the-art cycle-accurate simulators such as gem5 are commonly used. Despite its flexibility and relative accuracy, one significant drawback is its simulation speed. As addressed in [6], compared with translation-based simulators such as QEMU [4], gem5 is significantly slower. This was demonstrated in [20], which was claimed to be the largest cycle-accurate simulations ever conducted with research-driven gem5. As the authors stated, the benchmarks alone took multiple months to run, and even then some were still missing due to gem5-related issues or exceeding the time limit of the simulation. Despite the complexity and scale of their experiments, it is still evident that gem5 lacks scalability.

To test the slowdown of gem5 (version 22.1), we ran all the benchmarks in PolyBench-C (version 3.20). We cross-compiled the benchmarks into RISC-V binaries and executed them in three different environments. RISC-V was chosen as the primary ISA, and the reason for this is explained later in Section 3. The environments include **(i)** the native RISC-V chip, **(ii)** QEMU user-mode emulation with a custom instruction tracing plugin, and **(iii)** gem5. The RISC-V board we used was a StarFive VisionFive with 2 CPUs and 8GB of memory. The server on which we executed QEMU and gem5 has an AMD Ryzen 5 CPU and 16 GB of RAM. The configuration of gem5 is as follows: SE mode, 1 GHz RiscvO3CPU with 16GM DRAM with 50ns latency, 16kB L1i, 64kB Lid caches, and 256 kB L2 cache. As indicated by the results in Fig 1, the slowdown caused by gem5 is mainly in the rage of 100× and 900×, whereas on average, our plugin is only 5× to 10× slower than the baseline. As an example, it takes gem5 around 100 seconds to run the small *covariance* kernel, while the same benchmark completes in 4.3 seconds on a RISC-V board. This discrepancy highlights the scalability issues of using gem5 to execute large HPC applications or to perform parameter sweeps for assessing memory latency sensitivity. In contrast, our EDAN uses QEMU to trace the programs, which is easily an order of magnitude faster than gem5.

To generate DAGs for a program, one may propose to exploit memory traces or MPI traces, yet neither is sufficient in this scenario. A pure memory trace does not contain dependency information between memory accesses, and an MPI trace is too coarse-grained for exposing memory-level parallelism. Therefore, our approach is necessary as it allows us to accurately identify data dependencies between instructions in the most fine-grained manner.

## 2 Background

### 2.1 Execution DAG (eDAG)

An execution directed acyclic graph (eDAG) is a way to represent the data dependencies between computations. It is also named computation DAG (CDAG) in the literature [36, 37, 69]. In this work, we opted for the term eDAG to emphasize the dynamic nature of EDAN and the fact that graphs are generated by executing and tracing programs. Formally, an eDAG can be expressed as a directed graph $G = (V, E)$. $V$ represents the set of instructions in a program and edges $E \subseteq (V \times V)$ denote the set of directed edges defining the data dependencies between instructions [18, 50, 70]. Fig 2 provides an example of a trivial C program in which three variables `a`, `b`, and `c` are initialized, and added together to another variable `sum`. Assuming that the instruction trace we obtain through the execution of the said C program is represented by 5 lines of RISC-V assembly, its eDAG will have 5 vertices where vertex 0, 1, and 3 initialize the variables respectively. Edges exist, in this scenario, between vertex 2 and 0 as well as 2 and 1 as the instruction `add a3,a3,a4` depends

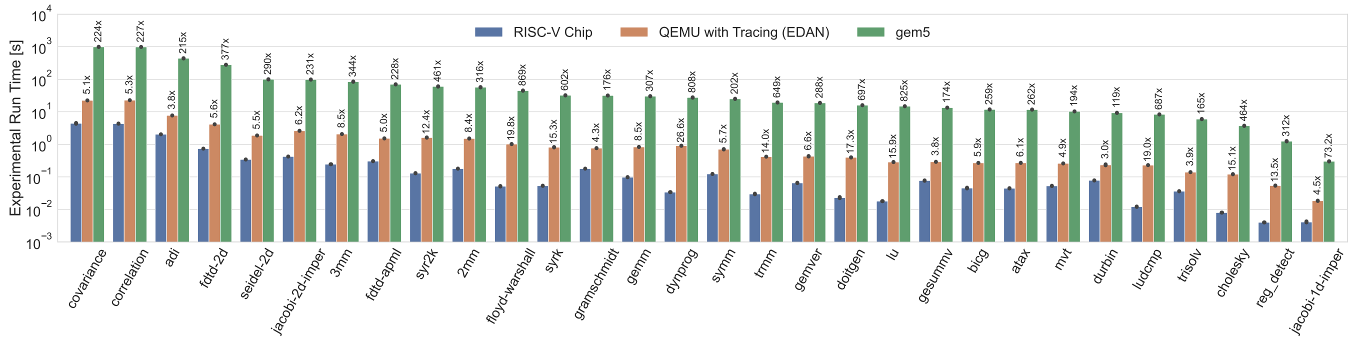


**Figure 1: Simulation time of Polybench kernels (small size) using QEMU emulation with instruction tracing (EDAN), and gem5 cycle-accurate simulation. Runtime on a RISC-V chip is used as the baseline for slowdown calculations.**

```c
int a = 1;
int b = 2;
int c = 3;
int sum = a + b + c;
```

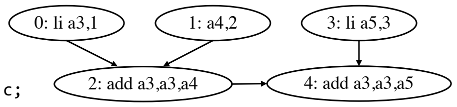


**Figure 2: A simple C program calculating the sum of 3 variables and its corresponding eDAG.**

on the values both in register a3 and a4. A similar argument can be applied to construct the dependencies between vertex 2, 3, and 4.

## 2.2 DAG-based Performance Analysis

The amount of work ($T_1$) refers to the total time needed to execute every instruction in the program by one processor [18]. Mathematically, $T_1 = \sum_{v \in V} t(v)$, where $t$ is a function that, given any vertex $v$, outputs the time it takes to execute $v$. If every $v$ has a unit cost, $T_1$ equals the total number of vertices in the eDAG. The depth of an eDAG ($T_\infty$), also referred to as span in parallel programming literature, is the shortest time required to execute all instructions when an unlimited number of processors are used. It is the aggregate execution time of vertices along the longest path, where the longest path is computed by weighting each vertex by its execution time [18]. Such a path is also called the *critical path* of the eDAG. The depth can be expressed as $T_\infty = \max_\pi T(\pi)$ where $\pi$ denotes a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $v_1$ and $v_k$ are input and output vertices respectively, and $T(\pi) = \sum_{v \in \pi} t(v)$. The degree of parallelism is the ratio between $T_1$ and $T_\infty$ and can be regarded as the average number of vertices that can be executed concurrently at each step along the critical path of an eDAG. Intuitively, a higher degree of parallelism indicates that on average, more tasks can be executed simultaneously, which can result in a faster overall execution time for programs. The *work law* claims $T_p \geq \frac{T_1}{p}$, where $p$ is the number of available processors and $T_p$ is the execution time of the program. The *span law* states that $T_p \geq T_\infty$. These two laws express that $T_p$ cannot be less than the average work done by each of the $p$ processors or the critical path through the eDAG. The work and span laws together define the lower bound of $T_p$ as $T_p \geq max\{\frac{T_1}{p}, T_\infty\}$. Given $T_1$ and $T_\infty$, $p$ processors, and a greedy scheduler, Brent's lemma states that the execution time of a program $T_p$ is upper bounded by $\frac{T_1 - T_\infty}{p} + T_\infty$ [11, 26].

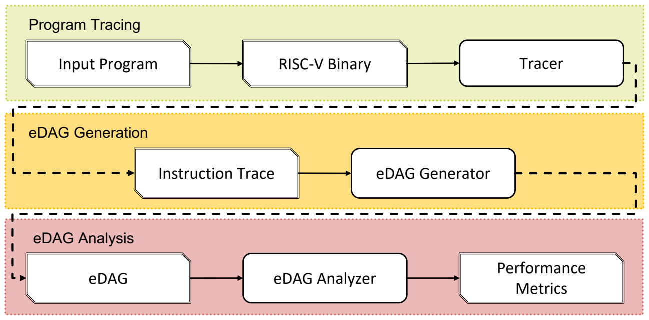


**Figure 3: High-level overview of the EDAN toolchain.**

## 3 eDAG Analyzer Toolchain

Fig 3 illustrates the overall structure and the elements involved in the workflow of the EDAN toolchain. As one can see from the colored blocks, it is composed of three main stages, whose respective functionalities are tracing programs, generating eDAGs based on the collected trace, and producing relevant performance metrics from eDAGs. The principal components and design choices in each stage are discussed in detail in the following sections.

We chose RISC-V as the target ISA, and the reason is two-fold. First, RISC-V is relatively simple [67], with fewer instructions to consider than more intricate ISAs such as x86 [31] and ARM64 [1]. RISC-V instructions are also less complex, in terms of their addressing modes and flag control. This greatly reduces the complexity of the instruction trace parser in the second stage as well as the time it takes to prototype the tool. Second, since the RISC-V project's inception [2], it has garnered significant interest from both academia and industry [21], leading to the development of numerous new extensions and hardware support [3, 63]. We hope to contribute to the thriving open-source RISC-V community and the ecosystem through the introduction of EDAN. The modular design of EDAN allows for easy incorporation of support for other ISAs by implementing their corresponding trace parsers and eDAG generators without affecting other parts of the toolchain.

## 3.1 Program Tracing

The primary goal of the first stage is to obtain a trace of every assembly instruction that has been executed in a program. To start, we take the source code of an arbitrary application and compile it to RISC-V binary. In order to achieve this, we primarily leveraged the RISC-V GNU Toolchain (GCC version 12.2.0) [57] as most users might not have access to hardware that is capable of support the RISC-V ISA.

```c
#define N 4
int __attribute__ ((noinline))
kernel(int *arr, int n)
{
  int i, sum = 0;
  // Perform summation
  for (i = 0; i < n; ++i)
      sum += arr[i];
  return sum;
}
```

**Figure 4: Example kernel in C that sums all elements in a given array.**

```
add a3,a0,a1
mv a0,zero
lw a4,0(a5);0x400800290
addi a5,a5,4
addw a0,a0,a4
bne a3,a5,-6
lw a4,0(a5);0x400800294
addi a5,a5,4
addw a0,a0,a4
bne a3,a5,-6
```

**Figure 5: Section of the trace generated from the summation kernel.**

*3.1.1 Tracer* Many tools can be employed to trace a program, including *perf* [64] or *gdb* [65]. Nonetheless, they are either too slow or do not allow the trace output to be customized easily. To this end, we made the decision to utilize the Tiny Code Generator (TCG) plugin in QEMU (version 7.2.91) user mode [5] as the core of EDAN's tracer. This approach has several benefits. Firstly, under user mode, QEMU is extremely fast as TCG translates target instructions and syscalls to be compatible with the host without having to emulate the OS kernel or the hardware. Secondly, TCG plugins are C programs that gain access to runtime information and interact with QEMU via APIs. Hence, by writing our own TCG plugin and modifying parts of the disassembler, we are able to easily tailor the output to our desired format and maximize the performance of program tracing. Lastly, unlike ISA-specific emulators such as rv8 [16] and banshee [56], similar TCG plugins can be attached to QEMU emulators with different target ISAs, enabling assembly traces of various ISAs to be collected.

Our tracer plugin provides users with the flexibility of specifying the names of particular functions that will be traced or will be excluded from tracing. This way, only instructions from the most crucial functions are recorded, while irrelevant function calls, such as those directed to the runtime library, are ignored. Not only does this ameliorate the slowdown of tracing, but it also mitigates the noise from nonessential functions and the possible impact it may have on later stages. Sometimes this requires adding flags such as `-g` or `-fno-inline` to the compiler, which ensures that the corresponding symbols can be accessed during tracing.

Fig 4 shows a code that traverses through an integer array `arr` of size `n` and returns the sum of all of its items. It will serve as a running example throughout the paper and be referred to as the summation kernel. Fig 5, on the other hand, presents one section of the trace that was obtained through the execution of the summation kernel. The instruction trace contains two columns of data separated by semicolons. The first column shows the assembly instruction, and the last column, which is not always present, denotes the virtual address of the data of a memory access operation, such as memory loads and stores. Data addresses are necessary for generating eDAGs and incorporating cache models into their analysis.

## 3.2 eDAG Generation

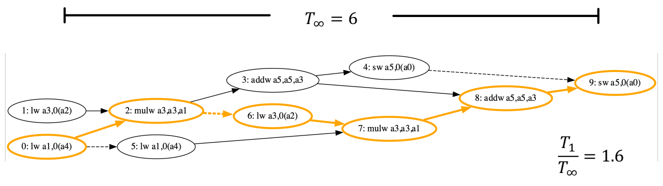
After program tracing, the next step is to parse the trace and generate an appropriate eDAG that captures the data dependencies between instructions. To achieve this, we developed a trace parser and eDAG generator written in Python.

The principal ideas behind the eDAG generator are presented by the Python-style pseudocode in Algorithm 1. As the program iterates through the trace file, it splits each line into the instruction itself and the data address (or addresses in some rare cases) it accesses. If an address is present, the cache model is used to simulate whether the given memory address is a cache hit. The instruction cost model determines the computation cost (e.g., number of CPU cycles) of each instruction, and it can be regarded as the function $t$ in Section 2.1. Both models are employed to obtain information that will be essential to computing the performance metrics from eDAGs. The core algorithm that establishes the dependencies between vertices is summarized by lines 12 to 17. This procedure ensures that all data dependencies can be correctly identified while constructing the eDAG. At the same time, it abstracts away all the ISA-specific functionalities into the function `generate_vertex()`, so that when more ISAs are incorporated into EDAN, it is the only component that needs to be implemented.

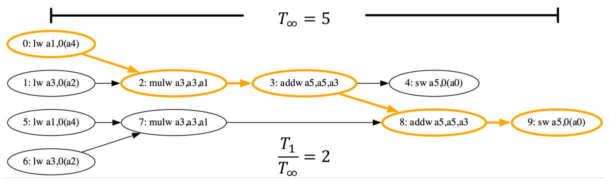
One potential restriction of applying the cache model to the memory addresses according to the sequential order in the trace is that when $N$ memory accesses are executed, there are in reality $N!$ ways to order them. Distinctive orderings will result in dissimilar cache miss rates. Therefore, all topological sortings of memory access vertices should be considered in theory. Nonetheless, that would be computationally intractable, and thus we decided to only follow one specific ordering of the memory accesses.

**Algorithm 1** Pseudocode for generating eDAGs from trace files

```
Input: Trace file $trace$, Cache model $cache$ with parameters $\theta$,
       Instruction cost model $t$ with parameters $\phi$
 1: Initialize new $eDAG$ object
 2: Initialize a dictionary $curr\_vs$
 3: for each $line$ in trace do
 4:   $insn, data\_addr = line.split(';')$
 5:   $v = generate\_vertex(insn, data\_addr)$
 6:   if $data\_addr$ is not None then
 7:     $v.cache\_hit = cache.get(data\_addr)$
 8:   $v.time = t.get\_cost(v)$
 9:   $eDAG.add(v)$
10:   $targets = v.targets$
11:   $dep\_vals = v.dep\_vals$
12:   for each $val$ in $dep\_vals$ do
13:     $dep\_v = curr\_vs[val]$
14:     if $dep\_v$ exists then
15:       $eDAG.add\_edge(dep\_v, v)$
16:   for each $target$ in $targets$ do
17:     $curr\_vs[target] = v$
18: return $eDAG$
```



(a) Example eDAG where non-true dependencies are dashed arrows.



(b) The same eDAG where only true dependencies are present.

**Figure 6: Removing non-true data dependencies can help reduce the depth of the eDAG and expose potential parallelism. A critical path in both graphs is highlighted.**

*3.2.1 Exposing Potential Parallelism* There are four categories of data dependencies, which include true dependencies (read-after-write dependencies or RAW for short), anti-dependencies (write-after-read dependencies or WAR for short), output dependencies (write-after-write dependencies or WAW for short), and input dependencies (read-after-read or RAR for short) [39]. We noticed that considering non-true data dependencies in eDAGs greatly hinders the discovery of potential instruction-level parallelism that is intrinsic to an application [49]. This can be primarily attributed to the fact that in a realistic microarchitecture, only a limited number of registers are available. Hence, false dependencies, especially WAW, can be seen as introduced by the register allocation algorithm in a compiler [40] as a means to cope with this constraint.

To demonstrate how this approach exposes potential instruction-level parallelism from the trace of a purely sequential program, we present Fig 6. The two subfigures show a segment of an eDAG generated from a matrix multiplication kernel. The difference is that in Fig 6a, both WAW and RAW dependencies are kept and WAW dependencies are denoted by dashed arrows, while in Fig 6b WAW dependencies are removed. Assuming that all vertices have unit cost, the work $T_1$ in both cases is equal to 10, yet in the first scenario, $T_\infty$ is 6 whereas $T_\infty$ is 5 in the second scenario. In this specific example, as vertex 6 no longer need to wait for vertex 2 to free register `a3`, all load instructions can be executed at the same time. Thus, by simply ignoring false data dependencies, we have increased the average degree of parallelism from 1.6 to 2.

One limitation of our current approach is that it cannot fully uncover the potential instruction-level parallelism when register spilling happens. To be more precise, depending on the register pressure of the ISA and the design of the compiler, the values of some variables will be spilled to the main memory and stored back throughout the execution of the program [14]. This process creates additional dependencies between instructions and decreases the maximum degree of parallelism the program is capable of achieving given enough registers. Our toolchain does not detect register spilling, and hence not all parallelism can be recovered.

Fig 7 demonstrates the eDAG generated from the summation kernel trace with a 4-item input array. The eDAG shows that register `a0` stores `sum`, and vertices on the right add each array item to the sum. On the left, register `a5` increments as the index into the array, serving as the address for the next item to load. Branch instruction vertices (e.g., 7, 11, 15) do not overwrite register or memory address values. As eDAG only considers data dependencies and ignores control flow dependencies, no other vertices depend on them.

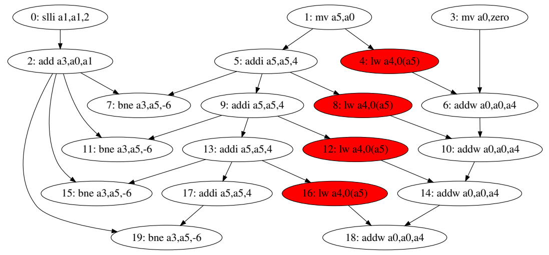


**Figure 7: eDAG generated from the trace of the summation kernel for n=4. Red vertices represent memory accesses, white vertices denote other instructions. Edges represent true dependencies.**

## 3.3 eDAG Analysis

To achieve one of the primary objectives of this work, which is to obtain performance metrics including the theoretical memory latency sensitivity of any given program, the generated eDAGs are passed to the eDAG analyzer. Before discussing the details of the metrics, we first define an appropriate cost model.

*3.3.1 Memory Cost Model* To mitigate the effects of the high latency and low bandwidth of memory accesses, CPUs employ two orthogonal techniques. First, multiple memory access instructions can be pipelined and executed in parallel [25]. This can increase the throughput of the system as long as the *memory-level parallelism* is high enough. Second, caches can reduce the number of memory accesses that need to access RAM, which improves the average latency as long as the program has good *locality* [8]. We extract a metric that takes into account memory-level parallelism and locality from our eDAGs. We assign each memory access that goes to RAM a constant access latency of $\alpha$ and we assume that $m$ memory accesses can be issued in parallel. This means that issuing $s$ memory accesses in parallel costs $\lceil \frac{s}{m} \rceil \alpha$. In contrast, for a chain of $s$ dependent accesses to RAM, the cost is $s\alpha$. In addition to the memory cost, we consider the total computational cost of non-memory-access operations (e.g., arithmetic, and cache access) in an eDAG to be a constant $C$ that is independent of $\alpha$ and proportional to the work. A vertex that accesses RAM (and is hence a cache miss) is a *memory access vertex.*

In general, we subdivide the eDAG into layers, which we define recursively: The first layer consists of memory access vertices that are not reachable from any other memory access vertices. The $i+1$-th layer consists of memory access vertices that are reachable from vertices in the $i$-th layer without going through another memory access vertex. The number of layers is the *memory depth* $\mathcal{D}$ and the total number of memory access vertices in the eDAG is the memory work $\mathcal{W}$. Let $\mathcal{W}_i$ be the number of memory access vertices in level

$i$. Based on these variables, the memory cost $M_{m,\alpha}$ is bounded by

$$\max\left(\mathcal{D}, \frac{\mathcal{W}}{m}\right)\alpha \leq M_{m,\alpha} \leq \left(\frac{\mathcal{W}-\mathcal{D}}{m} + \mathcal{D}\right)\alpha \quad . \tag{1}$$

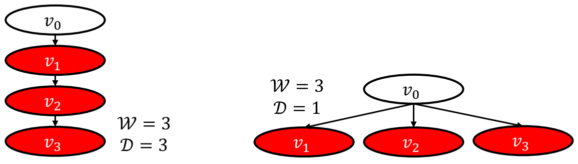
which can be obtained in similar reasoning as the work/span laws and Brent's lemma: For the lower bounds, notice that memory accesses that depend on each other must execute one after the other. Consider a path that contains the largest number of memory access vertices in the eDAG. Its length is $\mathcal{D}$, which yields the lower bound of $\mathcal{D}\alpha$. For the second lower bound, notice that at most $m$ memory accesses can occur in parallel. As there are $\mathcal{W}$ memory access vertices, the bound $\frac{\mathcal{W}}{m}$ follows. For the upper bound, observe that by definition of a layer, the next layer can execute as soon as the previous layer has finished and there are no memory dependencies within a layer, i.e., every memory access in a given layer can be issued in parallel. Hence, the memory cost to execute layer $i$ is $\left\lceil\frac{\mathcal{W}_i}{m}\right\rceil\alpha$ and the total memory cost is bounded by the sum $\sum_{i=1}^{\mathcal{D}}\left\lceil\frac{\mathcal{W}_i}{m}\right\rceil\alpha$ over the layers. Then, the inequality follows by using the rule $\lceil\frac{n}{m}\rceil = \lfloor\frac{n-1}{m}\rfloor + 1$, which holds for positive $n$ and $m$. To be more precise, Equation 1 describes the theoretical upper and lower bounds of the execution time of a program if its eDAG only contains memory access vertices while all other operations are ignored. Note that variables $\mathcal{D}$, $\mathcal{W}$, $C$, $M_{m,\alpha}$ and $T_{m,\alpha}$ are all functions of a given eDAG $G$. If $G$ is clear from the context, it is omitted from the expressions. Now, if we take into account the constant computation cost of non-memory-access vertices we can bound the total theoretical cost of the eDAG, $T_{m,\alpha}$, for a given $m$ and $\alpha$ as:

$$\max\left(\mathcal{D}, \frac{\mathcal{W}}{m}\right)\alpha + C \leq T_{m,\alpha} \leq \left(\frac{\mathcal{W}-\mathcal{D}}{m} + \mathcal{D}\right)\alpha + C \quad . \tag{2}$$

For simplicity, this cost model ignores the interactions between memory access vertices and other instructions. Nevertheless, it still provides us with an effective estimation of the impact of both the memory access latency and the number of available memory issue slots without having to develop a complex model that considers all the intricacies of the underlying architecture of the machine on which the program is run.

*3.3.2 Memory Latency Sensitivity* In mathematics, sensitivity analysis (SA) aims to investigate how the set of $N$ input variables $x = \{x_1, x_2, \ldots x_N\}$ influence the output $y = \{y_1, y_2, \ldots, y_D\}$ of a function $y = g(x)$ where $g : \mathbb{R}^N \to \mathbb{R}^D$ [10, 61]. There are two general approaches to conducting SA: local and global. Although global SA provides a more comprehensive view of the effect of each parameter in $x$, local SA is easy to implement and is not as computationally demanding. However, local SA has its limitations. For instance, when the model is nonlinear, the results produced can be heavily biased [59, 60]. Assuming that function $g$ is differentiable, derivative-based local SA can be performed by taking the partial derivative of $y$ with respect to the $i$-th input $x_i$ and expressed as $S_i = \frac{\partial y}{\partial x_i}\Big|_{x_0}$, where $S_i$ is the sensitivity measure of $x_i$ and $x^0 \in \mathbb{R}^n$ is the fixed point at which the derivative is evaluated [9, 53].

To derive memory latency sensitivity based on eDAGs, we refer to the theory regarding derivative-based local SA. In essence, we can take one of the bounds of $T_{m,\alpha}$, and compute its partial derivative with respect to $\alpha$. It is evident that the derivative expresses the quantity of how much $T_{m,\alpha}$ changes as the memory access latency varies, and can be utilized to directly gauge the memory latency sensitivity of an application. Moreover, considering that the model is linear, we also minimize the influence of potential bias from local SA [55, 60]. Since we are interested in the worst-case performance of an application, we opted for the upper bound of $T_{m,\alpha}$, and define the *absolute memory latency sensitivity* of an eDAG $\lambda$ as



**(a) Example eDAG of a latency-sensitive application.** **(b) Example eDAG of a latency-insensitive application.**

**Figure 8: eDAGs generated from a latency-sensitive vs. latency-insensitive application, red vertices denote memory accesses.**

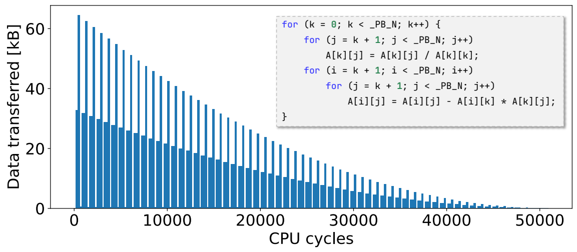
$$\lambda = \frac{\partial\left(\left(\frac{\mathcal{W}-\mathcal{D}}{m} + \mathcal{D}\right)\alpha + C\right)}{\partial\alpha} = \frac{\mathcal{W}-\mathcal{D}}{m} + \mathcal{D} \tag{3}$$

Fig 8 demonstrates the features in an eDAG that distinguishes a latency-sensitive application from a latency-insensitive one. Given the same amount of memory work $\mathcal{W}$, eDAG $G_1$ in Fig 8a should be more sensitive to memory latency due to the fact that it has all the memory access vertices clustered along the critical path, while eDAG $G_2$ in Fig 8b should be more tolerant to it as it only has a memory depth of 1. Assuming $m = 3$ and $\alpha = 1$, if we increment $\alpha$ to 2, $T_{m,\alpha}(G_1)$ will be increased by 3 while $T_{m,\alpha}(G_2)$ will only be increased by 1. Now, if we limit $m$ to 1, the cost increase will be 3 for both. From this example, it can be seen that the effect of depth on the overall memory latency sensitivity is constrained by the number of available memory issue slots. This characteristic is summarized perfectly by Equation 3. After re-arranging, we have $\lambda = \frac{1}{m}\mathcal{W} + (1 - \frac{1}{m})\mathcal{D}$, which signifies that given a fixed $m$ and $\mathcal{W}$, $\lambda$ grows with $\mathcal{D}$. If $\mathcal{W}$ and $\mathcal{D}$ stay constant, $m$ can be interpreted as the variable that controls what proportions $\mathcal{W}$ and $\mathcal{D}$ should be counted towards the total computation cost. The larger $m$ becomes, the more weight is given to $\mathcal{D}$ and vice-versa.

Despite being a useful metric, $\lambda$ on its own does not fully describe how an application's performance will be affected in comparison to a baseline. If an application is already slower than other programs, introducing additional memory access latency could lead to a comparatively larger decrease in performance on an absolute scale. However, the slowdown may not be as significant relative to its baseline performance. Conversely, adding memory latency to a fast program may only result in a minor increase in execution time on an absolute scale, but the relative impact on performance could be substantial. To this end, we formulate the *relative memory latency sensitivity* $\Lambda$ of an eDAG as

$$\Lambda = \frac{\lambda}{\lambda\alpha_0 + C} \tag{4}$$

where $\alpha_0$ is the baseline latency of memory access operations. Intuitively, $\Lambda$ represents the relative performance change of an application with respect to a specific baseline. Unlike $\lambda$, it produces a normalized metric between 0 and 1, and by taking into account $C$, $\Lambda$ implicitly shows the percentage of a program's total computation cost that is attributed to memory accesses.



```
for (k = 0; k < _PB_N; k++) {
    for (j = k + 1; j < _PB_N; j++)
        A[k][j] = A[k][j] / A[k][k];
    for (i = k + 1; i < _PB_N; i++)
        for (j = k + 1; j < _PB_N; j++)
            A[i][j] = A[i][j] - A[i][k] * A[k][j];
}
```

**Figure 9: Data movement over time of the *lu* kernel. The dataset size is 64. No cache model is used. Memory access instructions take 200 cycles, other instructions have unit costs. $\tau$ is set to 1 cycle.**

*3.3.3 Bandwidth Utilization* In addition to memory latency sensitivity, one can also approximate a program's average bandwidth utilization and visualize its data movement over time with the help of eDAGs. To do so, we first formulate the critical path length $T_\infty$ exactly as described in Section 2.2, and $w(v)$ as the amount of data moved between the CPU and the main memory in bytes when $v$ is processed. Then, under the assumption of a greedy scheduler and that an infinite number of instructions can be performed in parallel, the *average bandwidth utilization* $B$ can be expressed as

$$B = \frac{\sum_{v \in V} w(v)}{T_\infty} \tag{5}$$

Note that $B$ should be regarded as a reference to the theoretical maximum average bandwidth that can be achieved rather than an estimate of the actual bandwidth usage.

We then define $S(v)$, and $F(v)$ as the start time and finish time of vertex $v$ respectively. Given an eDAG $G = (V, E)$, $S(v)$ and $F(v)$ can be calculated as follows

$$S(v) = \begin{cases} 0 & \text{, if } v \in I \\ \max\{F(u) | (u, v) \in E\} & \text{, otherwise} \end{cases} \tag{6}$$

$$F(v) = S(v) + t(v) \tag{7}$$

where $I$ is the set of input vertices of $G$ (i.e. vertices whose in-degree is 0), and $t$ is a predefined function that outputs the execution time of $v$. Now, we can stratify the eDAG into $\lceil \frac{T_\infty}{\tau} \rceil$ phases given a specified time interval $\tau$, and the total data movement $U_i$ within phase $i$ can be expressed as $U_i = \sum_{v \in K} w(v)$ where $K = \{v | S(v) \leq \tau \cdot i \leq F(v)\}$ is a set containing all vertices that are being run in phase $i$. By assigning reasonable execution times to different types of instructions and adjusting the value of $\tau$, we can obtain sensible estimations of the data movement pattern of an application at various time resolutions.

Fig 9 illustrates the data movement plot generated from the trace of LU decomposition. We can clearly see that the peaks in the diagram delineate each iteration. Furthermore, the data movement pattern conforms with the intuition behind LU decomposition as the outer loop in the source code shows that the algorithm updates the upper triangular matrix from top to bottom, and the amount of transferred data decreases. Through this example, we showcase the possibility of using eDAGs to identify hidden data bursts in a program. More importantly, it is demonstrated that not only can eDAG be applied to theoretical analysis, but it is also capable of producing practical performance metrics.

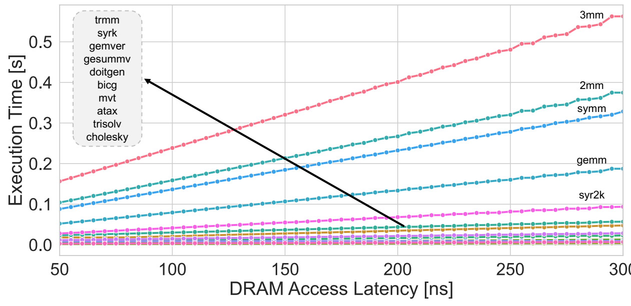


**Figure 10: Impact of increased memory access latency on the runtime of 15 PolyBench linear algebra benchmarks.**

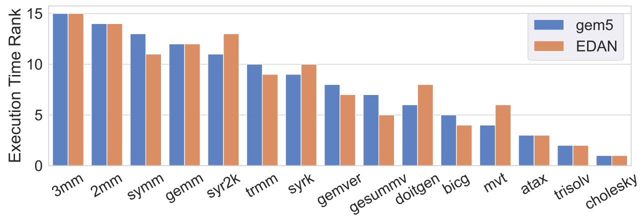


**Figure 11: Comparison of memory latency sensitivity rankings of benchmarks based on gem5 data and $\lambda$.**

## 4 Validation of EDAN

Despite the incorporation of an instruction cost model and the assignment of execution time to vertices, EDAN by no means provides a direct estimation of the actual runtime of a program or the slowdown incurred by additional memory latency. Thus, to validate the memory latency sensitivity metrics and to assess the efficacy of EDAN, another approach has to be taken. The technique we opted for involves measuring the performance degradation of various applications, ranking them based on the perceived impact of memory latency, and then comparing the applications' ranks to those acquired by analyzing their eDAGs. To gather data for the first step, we had to resort to gem5 as we lacked hardware that would easily allow artificial memory access delays to be injected.

Considering the significant latency overhead of gem5, we chose a set of linear algebra benchmarks with small dataset size from PolyBench to be evaluated. The configuration of gem5 is as follows: SE mode, 1 GHz RiscvO3CPU with 16GM DRAM with 50ns latency, 16kB L1i and 64kB Lid caches. For simplicity, we did not attempt to emulate a multi-node system with remote memory, as performing parameter sweeps in gem5 for numerous parameters and applications will be excessively time-consuming. Instead, we varied the DRAM latency for all memory access instructions from the baseline to 300$ns$ at 5$ns$ increment. We used PolyBench's internal time reporting functionality to measure only the time of the computation kernel so that the time taken for initialization and cleanup is excluded [23].

### 4.1 Validation of $\lambda$

In Fig 10, we display the runtime of 15 linear algebra benchmarks in gem5 plotted against increasing DRAM latencies. To generate the ranking, we calculated the average execution time for each across all tested latencies. We sorted them from highest to lowest, with the first kernel being the most latency sensitive. To produce the

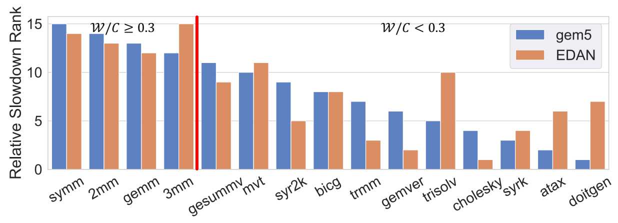


**Figure 12: Comparison of memory latency sensitivity rankings of benchmarks based on gem5 data and $\Lambda$.**

ranking with EDAN, we began by recording traces for the main computational kernels in the benchmarks, while disregarding extraneous functions like array initialization and cleanup. This ensured that the code section we traced would correspond accurately with the timing data provided by gem5. We generated the eDAG for each benchmark using the same parameters for the cache model as those used in gem5, in order to mirror the setting in gem5. We then calculated their respective $\lambda$ value with a value of 4 for $m$ and sorted them in descending order.

Fig 11 plots the comparison of rankings from gem5 and the $\lambda$ metric. As can be observed, 6 out of 15 benchmarks' rankings match perfectly with the ground truth produced by gem5, which includes the 2 most latency-sensitive as well as the 3 least latency-sensitive kernels. For those that are misaligned, the ranks differ by a maximum of 2, and the average difference between the two rankings is only 0.93. The ability to precisely identify benchmarks on both ends of the latency sensitivity spectrum and having a small average discrepancy suggest that $\lambda$ can be employed as a reliable metric to compare the potential increase in execution time of multiple applications when additional memory latency is introduced. Additionally, we demonstrate the effectiveness of the EDAN toolchain in enhancing the productivity of performance engineers. It took around 24 hours to collect the runtime data for all the benchmarks in gem5, while the entire process only took less than an hour with the assistance of EDAN. It is important to note that the value of $\lambda$ does not correspond to the magnitude of the execution time increase. In other words, unlike the slopes of the trendlines in Fig 10, if $\lambda_a$ from application $A$ is twice as large as $\lambda_b$ from application $B$, it does not necessarily indicate the runtime increase of $A$ will be twice that of $B$ in most cases.

## 4.2 Validation of $\Lambda$

To test the validity of $\Lambda$, we use the same data collected with gem5. Following the same methodology outlined in the previous section, we determined the relative slowdown of each benchmark's runtime when compared to its baseline (i.e. 50$ns$ DRAM latency) across all DRAM latencies. We ranked the benchmarks accordingly based on the average relative slowdown. For all benchmarks, we chose $\alpha_0$ to be 50 and the total number of non-memory-access vertices in an eDAG to be $C$. We observed that, in this case, the actual values of $\alpha_0$ and $C$ only affect the magnitude of $\Lambda$, and do not alter the rankings of the benchmarks.

Fig 12 presents the comparison of ranks based on the two approaches. However, the results here are significantly poorer than those in Fig 11. Specifically, only one ranking based on $\Lambda$ conforms to the ground truth, and the average discrepancy is 2.67. Nevertheless, we noticed that albeit not perfectly, EDAN predicted the top 4 most latency-sensitive benchmarks based on relative slowdown using $\Lambda$. Therefore, we sought to investigate the circumstances under which $\Lambda$ would give a reasonable estimate. To do so, we computed the value of $\frac{\mathcal{W}}{C}$, which denotes the ratio between memory work and the number of non-memory-access instructions. We discovered that the top 4 kernels all have a $\frac{\mathcal{W}}{C}$ ratio larger than 0.3. Based on this finding, it can be extrapolated that in order for $\Lambda$ to provide a sensible estimate, $\frac{\mathcal{W}}{C}$ needs to be above a certain threshold. This can be attributed to the fact that our metric does not accurately model the cost of non-memory access instructions. It overlooks the interactions between memory access vertices and all other instructions, making it impossible to know when computations and memory accesses overlap or depend on each other. Since the value of $C$ cannot be correctly computed, it follows that as the proportion of memory access vertices becomes smaller, the larger the deviation between the calculated $\Lambda$ and its actual value. Despite its weakness, $\Lambda$ is still a valuable metric for identifying memory-intensive benchmarks that could benefit from performance optimization strategies such as caching or prefetching.

## 5 Case Studies of Benchmarks and Applications

Now that we have validated our model and assessed the strengths and weaknesses of EDAN, we will proceed to apply it to a set of applications and benchmarks as case studies. This will enable us to gain an in-depth understanding of their potential memory-level parallelism and latency sensitivity.

### 5.1 PolyBench-C Suite

Although some experiments have already been performed on PolyBench in the previous section, analyzing the memory work $\mathcal{W}$ and memory depth of $\mathcal{D}$ of individual benchmarks can still provide further insight into memory-level parallelism, making it the first set of benchmarks to be examined. We varied the input data size $N$ for linear algebra benchmarks and investigated its impact on $\mathcal{W}$ and $\mathcal{D}$ of their eDAGs. The effect of $N$ on $\mathcal{W}$ is relatively uninformative, the relationship between them can simply be characterized by polynomial functions with different degrees according to the algorithms [37]. On the other hand, the connection between $N$ and $\mathcal{D}$ is more compelling. Fig 13 plots the values of $\mathcal{D}$ against $N$, and it can be seen that 8 out of 15 benchmarks have a constant memory depth despite a changing $N$. We attempted to categorize the benchmarks by the types of algorithms they perform, yet it was unsuccessful since algorithms that should belong to the same category, such as *trmm* and *2mm*, exhibit different behaviors for $\mathcal{D}$.

Upon closer inspection, we made the following discovery: **Data-oblivious applications should always have constant memory depths under the assumptions of an ideal architecture.** This is due to the fact that, by definition, in a data-oblivious application, both the memory access pattern and the control flow, are independent of the data itself [47]. Thus, memory loads that depend on each other, such as those found in pointer chasing, should not occur. If infinite registers are available to store all values, the longest chain of dependent memory accesses would be first loading a value from memory and storing it back after all dependent operations

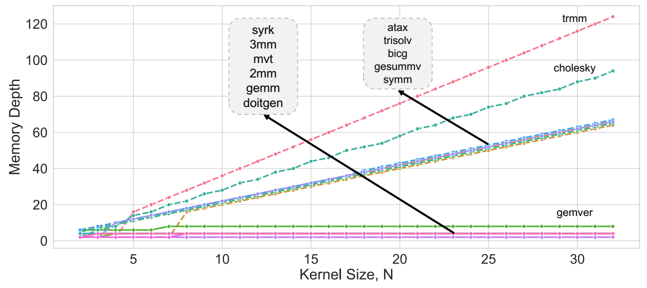


**Figure 13: Impact of data sizes on the memory depth $\mathcal{D}$ of PolyBench linear algebra benchmarks. Cache models were not used.**

have been executed, which makes the memory depth constant. An example would be the eDAG of the summation kernel in Fig 7. We can deduce visually that there is only one memory access vertex along the critical path regardless of the input size. Intuitively, it signifies that all elements from the array can be loaded at the same time, which shows significant memory-level parallelism. However, if we inspect it from a traditional work and depth perspective, this is all hidden as the depth grows linearly with the input size. In essence, through $\mathcal{D}$ and $\mathcal{W}$, we expose the potential memory-level parallelism in a program that is otherwise not easily detectable.

```
/* trmm: B := alpha*A'*B, A triangular */
for (i = 1; i < _PB_NI; i++)
    for (j = 0; j < _PB_NI; j++)
        for (k = 0; k < i; k++)
            B[i][j] += alpha * A[i][k] * B[j][k];
```

**Figure 14: Section of source code from *trmm*.**

Despite being data-oblivious, around half of the tested benchmarks still have a linear memory depth, which is primarily caused by register spilling as discussed in Section 3.2.1. To demonstrate this, we present in Fig 14 a section of the source code from *trmm*. In this case, *trmm* has the fastest-growing memory depth among all benchmarks. From its source code, we see that the compiler is unable to keep each `B[i][j]` in a designated register as there are too many distinct values loaded between its first and last access. For instance, when the kernel size is 4, 15 unique values are loaded from memory between the first and last access of `B[1][0]`. Since the compiler does not keep all of them in registers, the value `B[1][0]` will be "spilled" back to memory. This, in turn, creates extraneous dependencies between loads and stores.

## 5.2 HPCG

HPCG (High-Performance Conjugate Gradient) is a benchmark for ranking computer systems, and it centers around solving a large sparse linear system with preconditioned conjugate gradient (PCG) method [27, 66]. The benchmark consists of two main phases: the setup phase and the PCG iteration phase. The setup phase constructs the sparse matrix and the multigrid hierarchy, while the PCG iteration phase performs multiple iterations of the PCG algorithm to compute an approximate solution of the equation. The version of the program was 3.1.

To analyze the program's performance, we focused on the `CG` function in the PCG iteration phase and ignored the setup phase entirely. We chose a data size of 16 and an iteration number of 50. Tracing took approximately 35 seconds and produced a file of 5.5$GB$, containing over 210 million lines of instructions. The trace file was processed on a server with Intel Xeon X7550 CPUs, 1 TB of memory, and a PERC H700 hard disk. It took around 7 hours to generate and analyze the eDAG. We collected performance metrics from the eDAG with various cache configurations. $m$ and $\alpha_0$ were set to be 4 and 1 respectively, and the value of $C$ is the number of non-memory access vertices. We specified the cost of memory accesses to be 200 cycles, while all other instructions had a unit cost. The cache model we used was a write-through 2-way associative L1 cache with 64 bytes cache line and LRU as the eviction strategy. The results are summarized in Table 1.

| Cache Size | $\mathcal{W}$ | $\mathcal{D}$ | $\lambda$ | $\Lambda$ | B [GB/s] |
|---|---|---|---|---|---|
| No Cache | 106151255 | 73703 | 26593091 | 0.1462 | 46.5 |
| 32 kB | 11200012 (89.4%) | 45102 (38.8%) | 2833830 (89.3%) | 0.0112 (92.3%) | 8.1 |
| 64 kB | 10833505 (89.8%) | 43502 (41.0%) | 2741003 (89.7%) | 0.0108 (92.6%) | 8.1 |

**Table 1: Impact of cache sizes on the performance metrics in HPCG. The numbers in parenthesis show the percentage reduction compared to the baseline.**

One can see from the data that, in this specific scenario, caching plays a significant role in mitigating the memory latency sensitivity of HPCG. Compared with the baseline in which no cache was available, we see a reduction of around 90% for $\mathcal{W}$ when 32kB of cache is used, which, as one may expect, results in a substantial decrease in both $\lambda$, $\Lambda$, and the average bandwidth utilization $B$. However, increasing the cache size further leads to diminishing returns as doubling the cache size does not yield a noticeable improvement in performance metrics. This can be explained by the fact that a small dataset was used, which could likely fit within the cache, at which point, only the unavoidable cold misses remain.

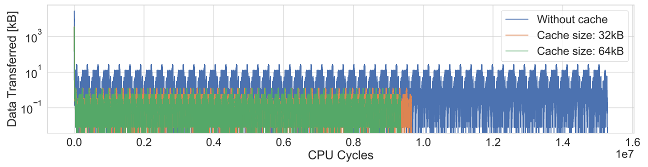


**Figure 15: Data movement over time of HPCG with different cache sizes. $\tau$ was set to 100 cycles.**

We visualize the data movement over time of the three configurations in Fig 15. The pattern exhibited in the plot adheres to our intuitive understanding of the algorithm as a large amount of data is loaded at the start, and the repetitive small bursts of data movement coincide with each PCG iteration. There are 50 peaks in the plot, which matches perfectly with the number of iterations we defined. Additionally, the impact of the cache is also visible as both the height and width of the orange and green lines are shorter compared with the baseline.

## 5.3 LULESH 2.0

LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) 2.0 is a proxy application that simulates the effect of a blast wave in a physical domain through time-stepping followed by calculation of the time constraint. The code uses an unstructured hexahedral mesh with two centerings, where the element centering stores thermodynamic variables, whereas the nodal centering stores kinematic values. The algorithm consists of two major steps: advancing the node quantities, followed by advancing the element quantities [33, 34]. To better understand LULESH, we traced its kernel function `LagrangeLeapFrog` with a data size of 1000 and an iteration number of 10. The tracing process took 3.8 seconds and produced a 1.2GB file with 49 million lines of instructions. It was then processed on the same server as described in the previous section and computed the performance metrics with an identical set of parameters. The results are presented in Table 2.

| Cache Size | $\mathcal{W}$ | $\mathcal{D}$ | $\lambda$ | $\Lambda$ | B [GB/s] |
|---|---|---|---|---|---|
| No Cache | 18852125 | 53776 | 4753363 | 0.1370 | 13.6 |
| 32 kB | 5389537 (71.4%) | 13083 (75.7%) | 1357197 (71.4%) | 0.0303 (77.9%) | 15.8 |
| 64 kB | 5279800 (72.0%) | 13055 (75.7%) | 1329741 (72.0%) | 0.0296 (78.4%) | 15.5 |

**Table 2: Impact of cache sizes on the performance metrics in LULESH.**

Compared with the data from HPCG, caching helps mitigate the memory latency sensitivity of LULESH in a similar fashion, as both $\mathcal{W}$ and $\mathcal{D}$ are decreased by more than 70% relative to the baseline. One difference is that the majority of memory vertices are removed from the critical path, resulting in a significant reduction in $\mathcal{D}$. Hence, the critical path length $T_\infty$ is also much shorter, which leads to a slight boost in $B$.

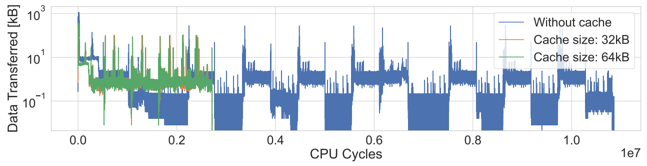


**Figure 16: Data movement over time of LULESH with different cache sizes. $\tau$ was set to 100 cycles.**

Fig 16 visualizes the data movement pattern of the computational kernel in LULESH, revealing its behavior during execution. The peaks in the plot indicate the start of a new time step, while the flat sections in between correspond to the calculation of nodal forces and the advancement of element quantities [33].

## 6 Related Work

***Performance Modeling Tools*** Computer architects often rely on simulators to evaluate the impact of architectural changes. Unsurprisingly, a plethora of simulators exist [6, 58, 62], capable of simulating architectures at various levels of details. Clearly, there is a trade-off between simulation accuracy and speed. While most of these tools simulate architecture components, others [62] rely on binary instrumentation [43] to trace events during execution and estimate latency. All simulation tools have in common that they force the user to do parameter sweeps to evaluate the impact of an architectural change on a specific application. This further increases the computational costs of relying on simulators to judge the impact of different parameters on performance. In this work, we rely on the gem5 simulator only to validate the accuracy of the predictions made by the proposed model. Constructing a graph-based representation of the program is a common approach in order to minimize the overhead of tracing and instrumentation [13, 54].

A possible solution is to construct models that abstract away most of the details of a computer, and instead, parameterize them using key performance metrics such as computational and memory bandwidth as in the original roofline model [68]. Naturally, such models are often extended with more parameters [29, 30, 44, 45]. While insightful, they only predict the performance of simple kernel, not an application made up of parts exposing different behaviors. Another issue is when modeling performance is finding suitable parameters to instantiate the model. Two other model families which target the differences in latencies for different memory accesses are the external memory model proposed in the balance principles for algorithm-architecture co-design [19] and variations on the red-blue pebble game [32].

This can be solved by combining modeling with tracing, i.e., the model is instantiated using a trace of an application. An example of this is the work by Cabezas and Püschel [13] in which kernel execution is traced, translated into an execution DAG, which is then scheduled according to micro-architectural constraints. This ultimately allows placing the kernel in a roofline plot. The methodology of this work is very similar to ours, however, we do not rely on the LLVM toolchain but directly on RISC-V binaries, and the model we instantiate from the eDAG is not a variant of the roofline model. Instead, we combine this approach with our own model which is inspired by the work-span model [7], which allows us to reach conclusions about the parallelism in memory accesses.

***Memory Latency Sensitivity Analysis*** The prior work on memory latency sensitivity analysis of workloads can be split into two categories: offline and online analysis. In offline analysis [13, 15, 20, 38, 48] the goal is to learn more about specific workloads or and how these workloads will react to changes in the machines they are executed on. While some offline analysis work [20, 48] relies on architectural simulation, others [13, 15] use a trace-based approach similar to this work, but either limit the user to a specific compiler toolchain [13] or do not offer a global view of the critical path in the application [15]. In online analysis, the goal is to improve the performance of a system by changing parameters on the fly [12, 35].

## 7 Limitations and Future Work

EDAN is capable of efficiently producing performance metrics for a wide range of programs. Nonetheless, as a purely experimental tool, it has a few notable limitations.

The drawbacks of the current memory cost model has already been uncovered in Section 4. The lack of a more accurate CPU and scheduler model causes EDAN to mispredict the relative computation cost of non-memory access instructions, which in turn reduces the accuracy of $\Lambda$. Developing a more comprehensive model will undoubtedly help ameliorate this discrepancy. However, this would likely introduce more computation overhead in the toolchain, and undermine the simplicity and efficiency of the current model.

As discussed extensively in Sections 3.2.1 and 5.1, EDAN is constrained both by the compiler and the underlying architecture to fully expose the memory-level parallelism intrinsic to a program. Techniques such as register spilling cause extraneous dependencies between memory accesses and greatly hinder the discovery of memory-level parallelism. Consequently, it would be beneficial to explore the possibility of extending compilers and emulators to enable the generation and execution of code with an unlimited number of registers, which could unlock EDAN's full potential.

Parallel programs are not yet supported by EDAN due to the sheer complexity of determining data dependencies in the presence of atomic operations, synchronization primitives, cache coherence protocols as well as message passing. These paradigms involve intricate interactions between multiple threads and processes that create convoluted data dependencies that cannot be easily inferred from program execution traces. Therefore, implementing support for parallel programs in EDAN would require significant algorithmic and structural changes to the existing toolchain.

Since EDAN relies heavily on the execution trace, it is vulnerable to input that only triggers a particular execution path, potentially generating misleading results. Moreover, the size of input data can also impact the outcome of performance analysis depending on the compiler. Therefore, to ensure accurate and generalized results, it would be beneficial to vary the input of a program.

Although EDAN is much more efficient compared to cycle-accurate simulators, its scalability can still be improved. One possible method is to store and parse traces in a binary format. This would reduce both the storage requirements and the computation needed for eDAG analysis. Additionally, EDAN should employ multiple processes so as to maximize the processing speed of large graphs.

Currently, EDAN relies on GCC with standard extensions (i.e., MAFD) to generate binaries for the riscv64 ISA. It would be valuable to explore the impact of using different compilers and riscv64 extensions, (e.g., vector extension), on program eDAGs. By doing so, we can broaden our understanding and generalize our findings.

## 8 Conclusion

In this work, we present EDAN, a novel experimental toolchain that exploits the execution DAG generated from the runtime trace of a sequential program to calculate theoretical performance metrics, such as memory latency sensitivity and average bandwidth utilization. To complement the toolchain, we developed a simple yet powerful memory cost model inspired by Brent's theorem and derivative-based sensitivity analysis. Based on this model, we derived two metrics $\lambda$ and $\Lambda$, which can be utilized to efficiently quantify and rank the memory latency sensitivity of applications.

By comparing our theoretical metrics with the experimental data collected from gem5, we tested the effectiveness of EDAN and understood the limitations of our model. Case studies were then conducted on several HPC benchmarks and applications, which include PolyBench, HPCG, and LULESH. Through the analysis of the performance metrics, we gained a deeper insight into the memory-level parallelism of an application, and more importantly, we demonstrate the practicality of EDAN in analyzing real-life applications.

As latency continues to increase in modern networks, efficient identification of application latency sensitivity is becoming increasingly crucial. With the development of EDAN, we have provided a tool to aid in this area, and we hope to inspire further advancements in this field.

## References

[1] ARM Limited. 2022. *Arm Architecture Reference Manual for A-profile architecture.* ARM Ltd. https://developer.arm.com/documentation/ddi0487/latest/

[2] Krste Asanović and David A. Patterson. 2014. *Instruction Sets Should Be Free: The Case For RISC-V.* Technical Report UCB/EECS-2014-146. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[3] MohammadHossein AskariHemmat, Theo Dupuis, Yoan Fournier, Nizar El Zarif, Matheus Cavalcante, Matteo Perotti, Frank Gurkaynak, Luca Benini, Francois Leduc-Primeau, Yvon Savaria, and Jean-Pierre David. 2023. Quark: An Integer RISC-V Vector Processor for Sub-Byte Quantized DNN Inference. arXiv:2302.05996 [cs.AR]

[4] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[5] Alex Bennée, Peter Maydell, Paolo Bonzini, and Christoph Müllner. 2022. qemu. https://github.com/qemu/qemu/blob/v7.2.0/docs/devel/tcg-plugins.rst.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[7] Guy E Blelloch. 1996. Programming parallel algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

[8] Bob Boothe and Abhiram Ranade. 1992. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (Queensland, Australia) *(ISCA '92)*. Association for Computing Machinery, New York, NY, USA, 214–223. https://doi.org/10.1145/139669.139729

[9] Emanuele Borgonovo. 2008. Sensitivity Analysis of Model Output with Input Constraints: A Generalized Rationale for Local Methods. *Risk Analysis* 28, 3 (2008), 667–680. https://doi.org/10.1111/j.1539-6924.2008.01052.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1539-6924.2008.01052.x

[10] Emanuele Borgonovo and Elmar Plischke. 2016. Sensitivity analysis: A review of recent advances. *European Journal of Operational Research* 248, 3 (2016), 869–887. https://doi.org/10.1016/j.ejor.2015.06.032

[11] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (apr 1974), 201–206. https://doi.org/10.1145/321812.321815

[12] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. 2019. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the Fourteenth EuroSys Conference 2019.* 1–16.

[13] Victoria Caparrós Cabezas and Markus Püschel. 2014. Extending the roofline model: Bottleneck analysis with microarchitectural constraints. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 222–231.

[14] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. *Computer Languages* 6, 1 (1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

[15] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. 2015. Quantifying the performance impact of memory latency and bandwidth for big data workloads. In *2015 IEEE International Symposium on Workload Characterization.* IEEE, 213–224.

[16] M. Clark. 2017. rv 8 : a high performance RISC-V to x 86 binary translator.

[17] Marcin Copik, Marcin Chrapek, Alexandru Calotoiu, and Torsten Hoefler. 2022. Software Resource Disaggregation for HPC with Serverless Computing. https://htor.inf.ethz.ch/publications/img/2022_copik_serverless_hpc_report.pdf

[18] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

[19] Kent Czechowski, Casey Battaglino, Chris McClanahan, Aparna Chandramowlishwaran, and Richard W Vuduc. 2011. Balance Principles for Algorithm-Architecture Co-Design. *HotPar* 11 (2011), 9–9.

[20] Jens Domke, Emil Vatai, Balazs Gerofi, Yuetsu Kodama, Mohamed Wahib, Artur Podobas, Sparsh Mittal, Miquel Pericàs, Lingqi Zhang, Peng Chen, Aleksandr Drozd, and Satoshi Matsuoka. 2022. At the Locus of Performance: A Case Study in Enhancing CPUs with Copious 3D-Stacked Cache. https://doi.org/10.48550/ARXIV.2204.02235

[21] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. 2021. A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations. In *Proceedings of the 18th ACM International Conference on Computing Frontiers* (Virtual Event, Italy) *(CF '21)*. Association for Computing Machinery, New York, NY, USA, 12–20. https://doi.org/10.1145/3457388.3458657

[22] John D'Ambrosia. 2022. IEEE P802.3df™ Defines Architecture Holistically to Achieve 800 Gb/s and 1.6 Tb/s Ethernet. *IEEE Standards Association* (2022). https://standards.ieee.org/beyond-standards/ieee-p802-3df-defines-a-holistic-architectural-approach/

[23] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/polybench.html

[24] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 249–264.

[25] Michael Golden and Trevor N. Mudge. 1993. *Hardware Support for Hiding Cache Latency.* https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=9ccb22c1e276804247c1f581a78b9716d66c7a73

[26] John L. Gustafson. 2011. *Brent's Theorem.* Springer US, Boston, MA, 182–185. https://doi.org/10.1007/978-0-387-09766-4_80

[27] Michael Allen Heroux and Jack. Dongarra. 2013. Toward a new metric for ranking high performance computing systems. (6 2013). https://doi.org/10.2172/1089988

[28] Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyan Shen, Abdul Kabbani, Moray McLaren, and Steve Scott. 2023. Datacenter Ethernet and RDMA: Issues at Hyperscale. arXiv:2302.03337 [cs.NI]

[29] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. 2013. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters* 13, 1 (2013), 21–24.

[30] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. 2016. Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores. *IEEE Trans. Comput.* 66, 1 (2016), 52–58.

[31] Intel Corporation. 2022. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel Corporation. https://www.felixcloutier.com/x86/

[32] Hong Jia-Wei and Hsiang-Tsung Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. 326–333.

[33] I Karlin. 2012. LULESH Programming Model and Performance Ports Overview. (12 2012). https://doi.org/10.2172/1059462

[34] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973. 1–9 pages. https://asc.llnl.gov/sites/asc/files/2021-01/lulesh2.0_changes1.pdf

[35] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 65–76.

[36] Grzegorz Kwasniewski, Tal Ben-Nun, Lukas Gianinazzi, Alexandru Calotoiu, Timo Schneider, Alexandros Nikolaos Ziogas, Maciej Besta, and Torsten Hoefler. 2021. Pebbles, Graphs, and a Pinch of Combinatorics: Towards Tight I/O Lower Bounds for Statically Analyzable Programs. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 328–339. https://doi.org/10.1145/3409964.3461796

[37] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. arXiv:1908.09606 [cs.CC]

[38] Oliver Lenke, Richard Petri, Thomas Wild, and Andreas Herkersdorf. 2021. PEPERONI: Pre-Estimating the Performance of Near-Memory Integration. In *The International Symposium on Memory Systems*. 1–6.

[39] Zhen Li, Ali Jannesari, and Felix Wolf. 2013. Discovery of Potential Parallelism in Sequential Programs. In *2013 42nd International Conference on Parallel Processing*. 1004–1013. https://doi.org/10.1109/ICPP.2013.119

[40] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. 1999. Evaluation of Algorithms for Local Register Allocation. In *Compiler Construction*, Stefan Jähnichen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 137–152.

[41] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. *SIGARCH Comput. Archit. News* 37, 3 (jun 2009), 267–278. https://doi.org/10.1145/1555815.1555789

[42] Ling Liu, Wenqi Cao, Semih Sahin, Qi Zhang, Juhyun Bae, and Yanzhao Wu. 2019. Memory Disaggregation: Research Problems and Opportunities. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1664–1673. https://doi.org/10.1109/ICDCS.2019.00165

[43] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[44] Diogo Marques, Aleksandar Ilic, Zakhar A Matveev, and Leonel Sousa. 2020. Application-driven cache-aware roofline model. *Future Generation Computer Systems* 107 (2020), 257–273.

[45] Diogo Marques, Aleksandar Ilic, and Leonel Sousa. 2021. Mansard roofline model: Reinforcing the accuracy of the roofs. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 6, 2 (2021), 1–23.

[46] George Michelogiannakis, Benjamin Klenk, Brandon Cook, Min Yee Teh, Madeleine Glick, Larry Dennison, Keren Bergman, and John Shalf. 2022. A Case For Intra-Rack Resource Disaggregation in HPC. *ACM Trans. Archit. Code Optim.* 19, 2, Article 29 (mar 2022), 26 pages. https://doi.org/10.1145/3514245

[47] John C. Mitchell and Joe Zimmerman. 2014. Data-Oblivious Data Structures. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 25)*, Ernst W. Mayr and Natacha Portier (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 554–565. https://doi.org/10.4230/LIPIcs.STACS.2014.554

[48] Richard Murphy. 2007. On the effects of memory latency and bandwidth on supercomputer application performance. In *2007 IEEE 10th International Symposium on Workload Characterization*. IEEE, 35–43.

[49] Onur Mutlu. 2021. Out-of-Order Execution. https://safari.ethz.ch/digitaltechnik/spring2021/lib/exe/fetch.php?media=onur-digitaldesign_comparch-2021-lecture16-out-of-order-execution-beforelecture.pdf

[50] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. 2014. A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures. *Communications in Computational Physics* 15, 2 (2014), 285–329. https://doi.org/10.4208/cicp.110113.010813a

[51] Archit Patke, Haoran Qiu, Saurabh Jha, Srikumar Venugopal, Michele Gazzetti, Christian Pinto, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2022. Evaluating Hardware Memory Disaggregation under Delay and Contention. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1221–1227. https://doi.org/10.1109/IPDPSW55747.2022.00210

[52] Ivy Peng, Roger Pearce, and Maya Gokhale. 2020. On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 183–190. https://doi.org/10.1109/SBAC-PAD49847.2020.00034

[53] Francesca Pianosi, Keith Beven, Jim Freer, Jim W. Hall, Jonathan Rougier, David B. Stephenson, and Thorsten Wagener. 2016. Sensitivity analysis of environmental models: A systematic review with practical workflow. *Environmental Modelling & Software* 79 (2016), 214–232. https://doi.org/10.1016/j.envsoft.2016.02.008

[54] Brian Paul Railing. 2015. *Collecting and representing parallel programs with high performance instrumentation.* Ph. D. Dissertation. Georgia Institute of Technology.

[55] Patrick M. Reed, Antonia Hadjimichael, Keyvan Malek, Tina Karimi, Chris R. Vernon, Vivek Srikrishnan, Rohini S. Gupta, David F. Gold, Ben Lee, Klaus Keller, Travis B. Thurber, and Jennie S. Rice. 2022. *Addressing Uncertainty in Multisector Dynamics Research.* Zenodo. https://doi.org/10.5281/zenodo.6110623

[56] Samuel Riedel, Fabian Schuiki, Paul Scheffler, Florian Zaruba, and Luca Benini. 2021. Banshee: A Fast LLVM-Based RISC-V Binary Translator. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643546

[57] RISC-V Collaborative Project. Accessed 2023. RISC-V GNU Toolchain. https://github.com/riscv-collab/riscv-gnu-toolchain.

[58] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, et al. 2011. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review* 38, 4 (2011), 37–42.

[59] Andrea Saltelli. 1999. Sensitivity analysis: Could better methods be used? *Journal of Geophysical Research: Atmospheres* 104, D3 (1999), 3789–3793. https://doi.org/10.1029/1998JD100042 arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/1998JD100042

[60] Andrea Saltelli, Ksenia Aleksankina, William Becker, Pamela Fennell, Federico Ferretti, Niels Holst, Sushan Li, and Qiongli Wu. 2017. Why So Many Published Sensitivity Analyses Are False. A Systematic Review of Sensitivity Analysis Practices. arXiv:1711.11359 [stat.AP]

[61] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto. 2004. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models.* Wiley. https://books.google.ch/books?id=NsAVmohPNpQC

[62] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news* 41, 3 (2013), 475–486.

[63] Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2020. Indirection Stream Semantic Register Architecture for Efficient Sparse-Dense Linear Algebra. arXiv:2011.08070 [cs.AR]

[64] Anup Sharma and Davidlohr Bueso. 2023. Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial. Accessed: March 23, 2023.

[65] Richard Stallman, Roland Pesch, and Stan Shebs. 2010. Debugging with gdb. https://www.eecs.umich.edu/courses/eecs373/readings/Debugger.pdf

[66] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. 2018. Chapter 4 - Benchmarking. In *High Performance Computing*, Thomas Sterling, Matthew Anderson, and Maciej Brodowicz (Eds.). Morgan Kaufmann, Boston, 115–140. https://doi.org/10.1016/B978-0-12-420158-3.00004-6

[67] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[68] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[69] Xiaoyang Zhang, Junmin Xiao, and Guangming Tan. 2020. I/O Lower Bounds for Auto-tuning of Convolutions in CNNs. arXiv:2012.15667 [cs.LG]

[70] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. 2020. DAG Scheduling and Analysis on Multiprocessor Systems: Exploitation of Parallelism and Dependency. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. 128–140. https://doi.org/10.1109/RTSS49844.2020.00022