Investigating register and memory access intervals for use in a distributed microarchitecture

D.A. Morano, D.R. Kaeli
Northeastern University
dmorano, kaeli@ece.neu.edu

26th June 2006

Abstract

We explore several access intervals (measured in instructions) between register assignments and the subsequent uses of the same registers. Likewise, we also explore memory stores and the subsequent loads from the same memory addresses. Three different types of access intervals are defined and data are gathered on them through the simulated execution of benchmark programs. These data serve to provide insight about how a future distributed microarchitecture might take advantage of the register or memory access behavior in order to reduce the overhead of accessing centralized machine resources like a register file or the L1 memory cache.

1 Introduction

This paper explores the access intervals, as measured in dynamic instructions executed, between assignment to registers (writes) and their corresponding uses (reads), as well as the assignment to memory locations (denoted by a memory address) and their corresponding uses. We define three types of access intervals that we want to explore. These access intervals are defined similarly for both register and memory operations. We have gathered data associated with each of our three access intervals, for both register and memory operations, through the execution of general purpose sequential program codes. This work was substantially inspired by the prior work of Franklin et al. [3] in their investigation of register traffic for use in the context of the Multiscalar-like microarchitectures. Prior work on memory locality has been substantial including such work as Madison et al. [8], Verkamo et al. [19], and more recently that of Phalke et al. [20] with their Inter-Reference Gap model. Eeckhout and Bosschere have also investigated access intervals for registers (they didn't consider memory) [1] but their goal was different than ours. They attempted to distill the access interval characteristics of registers into work-load parameters that could be used to drive an hybrid analytical-statistical modeling technique for new microarchitectures.

The present work builds on and extends that of Franklin et al. and also to some extent on that of Eeckhout and Bosschere by providing additional information on register operand traffic that was not previously reported. We also extend the previous work of both Franklin et al., Eeckhout et al. and Phalke et al. by applying our same measurements for register traffic operations to memory traffic operations.

The rest of this paper is organized as follows. Section 2 gives some motivation for why we wanted to perform access interval characterization. Section 3 presents our definitions of the access intervals that we studied along with some significance of those intervals for microarchitectural design decisions. Section 4 presents our characterization results. The results are presented in two parts: 1) for the register operations and 2) for memory operations. We summarize the present work in Section 5.

2 Motivation for access interval characterization

As computer microarchitectures have increased in complexity and size, the need to access and update centralized microarchitectural resources has become increasing problematic. Access contention for centralized microarchitectural resources increases substantially as the machine is enhanced to perform a greater and greater number of operations in parallel. The goal of these microarchitectures is to generally increase program execution performance through the extraction of higher amounts of Instruction Level Parallelism (ILP) from substantially sequential programs. Generally, this is achieved by introducing multiple execution units into the microarchitecture where each unit can independently process a part of the single threaded program's instruction stream. However, these multiple execution units still all need access to a common set of architected machine registers as well as the architected memory of the computer. In addition to access contention for centralized resources, the routing complexity required to access and update those resources becomes an increasingly difficult implementation issue in the silicon layout.

We envision that future microarchitectures that include much larger numbers of execution resources than those of today will encounter two primary problems with respect to their physical implementation. These problems are:

1. increasingly difficult access to centralized resource (routing and congestion complexity)
2. the difficulty of interconnecting the numberous machine resources

Both of these problems are addressed through the use of a distributed microarchitecture, one which is physically distributed across either the silicon chip or across a multichip module.

Some example proposed microarchitectures that have explored the use of multiple execution units are the Multiscalar-like processors [13, 14], the SuperThreaded processor model [16], and the Parallel Execution Window (PEW) processor model [7]. Another proposed microarchitecture that that features multiple execution units that is also suitable for being physically distributed (like the Multiscalar processor above) is the Ultrascalar processor by Henry et al. [5]. This particular microarchitecture bears much resemblance to that of the Multiscalar process but is more fine-grained in its degree of parallelism

Other microarchitecture proposals such as the MultiCluster machine model by Farkas et al. [2] are also in this category. More recently, much work on trace processor microarchitecture models [11, 12, 18] have also explored multiple distributed execution units and their associated problems. Another example of a distributed microarchitecture is that of the Grid Processor described by Nagarajan et al. [10]. Further, microarchitectures have been proposed that feature both a large number of distributed execution units along with more generalized multipath execution [9, 17].

An example of providing distributed access to the architected register file was described by Jiser et al. in their paper on *Global Register Partitioning* [6]. However, their approach (and similar to that of the Multiscalar work and the Grid Processor work mentioned above) includes the use of the compiler to facilitate a total distributed machine model. We are interested in a more restricted design space where an existing ISA must be maintained such is the case with the R10000 or the Alpha processors.

Our primary goal is to investigate distributed microarchitectures (similar to many of the above examples) that feature very large instruction windows with multiple distributed execution units. We want to explore the feasibility of having in-flight operands (of both register or memory variety) that can bypass both the architected register file as well the traditional L1 data cache and the load-store queue (LSQ) for subsequent uses of those operands. Although rudimentary register operand bypassing of the architected register file has been done for a long time [15], it has not been expanded into the realm of large and spatially distributed microarchitectures. Also, as the snooping of the familiar load-store queue can be thought of as bypassing (if there is a hit) the L1 data cache, it itself simply becomes the next centralized resource for which access contention becomes the problem. We want to investigate the feasibility of having either register or memory operands bypass any centralized machine resource entirely. Two general mechanisms can be envisioned to facilitate this type of operand bypass. One such mechanism is the direct forwarding of a generated operand from the source instruction to the corresponding sink instructions. This can generally only occur when both the source of the operand and the instruction sinks of the same operand are both located within the instruction window of the processor simultaneously. This prompts the question as to how often this can be expected to occur in general purpose programs. The second general mechanism to facilitate operand bypass is the use of buffering or caching within the execution window of the processor. More specifically, this would constitute either buffering or caching of operands at some spatial location between operand sourcing instructions and the corresponding sink instructions for the same operands. It is the idea of buffering intermediate register and memory operands and having them available (cached) for use by subsequent instructions (physically distributed or separated) that most motivates this present work.

3 Definitions and significance of intervals

There are several possible intervals that can be defined with regard to writes to a variable (whether a register or a memory location) and the corresponding reads of the same variable. We only explore three of these in this present work since other possible intervals do not have as important significance as the three types of intervals that we have selected. We first provide some definitions for the intervals that we have explored and then give some possible significance to those intervals for microarchitectural design purposes.

3.1 Definitions of intervals

We define all intervals in terms of the definition (hereafter referred to as a *def*) of a *variable instance* and the corresponding subsequent *use* of the same instance of the variable. A write to any variable always constitutes a def of a new variable instance. A read to a variable constitutes a use of the associated variable instance. For our purposes, reads and writes occur on variables while defs and uses occur for variable instances. Variable instances are uniquely identified by both their variable address and their associated def (as identified in dynamic program ordered time). For registers, the address is generally just its name. For memory variables, the address is that of the memory location itself. As alluded to already, a write to a variable destroys the previous instance of the variable while simultaneously creating a new instance of that variable. This idea of variable instance is the same as that of Franklin et al. [3]. A write to a variable is also assumed to constitute a def of a new variable instance even if the value assigned to the variable is the same as that which it had already. Uses of variable instances occur when the associated variable is read for any architected reason, whether this is explicit or implicit to the execution of an instruction.

We explore the following three types of access intervals on both registers and memory locations :

- *access-use*
- *useful-lifetime* (or *def-last-use*)
- *def-use*

The *access-use* interval is defined as the number of dynamic instructions from a read of a variable to the closest of a preceding read or write to the same variable. An access-use interval is therefore a property of a read of a variable. Note that there may be many uses of the same instance of a variable. Each read of a variable will therefore usually have a different access-use interval associated with it since the number of dynamic instructions from the def of the variable instance to the current use is usually different. Two uses may have the same access-use interval when, for example, each is associated with an input operand to a single instruction. Our definition for the access-use interval is the same as the *inter-reference gap* from the work of Phalke et al. [20]. Franklin et al. [3] did not consider this type of access interval in their work.

The *useful-lifetime* is defined as the number of dynamic instructions between the write of a variable and the *last* read of the same variable before the subsequent write to the same variable. This interval is also sometimes referred to as the *def-last-use* interval since only the last use of the variable instance is taken into account when determining this interval. This interval is therefore a property of the write to a variable. Each write of a variable therefore only has one useful-lifetime interval associated with it. If there are, for example, three uses following the def of a variable instance, only the last such use is used to determine the useful-lifetime of the variable instance. Note that the use-lifetime of a variable instance can have a value of zero. The term *useful-lifetime* is taken from Franklin [3]. This term is generally synonymous with the *liveness* of a variable instance [4]. It should be noted that the term *lifetime* is often confused with that of useful-lifetime. For our purposes, we define the term lifetime to refer to the interval from a def of a variable instance (write to a variable) to the subsequent write of the same variable (thus creating a new variable instance). This is generally different than the def-last-use interval for the same variable instance. This definition of lifetime is the same as that used previously by Franklin et al. [3]. We do not further explore lifetime intervals in this present work.

Finally, the *def-use* interval is the number of dynamic instructions from the read of a variable to the preceding write of the same variable. As the case was with the access-use interval, the def-use interval is a

Table 1: *Simple code example illustrating the different types of access intervals.* These machine parameters are used for all simulations.

label	instruction	event	intervals determined
I1	`r1 <= c`	def(r1)	
I2	`r2 <= c`	def(r2)	
I3	`r1 <= r2 + c`	def(r1), use(r2)	access-use(r2)=1, def-use(r2)=1, useful-lifetime(r1)=0
I4	`r2 <= r1 + c`	def(r2), use(r1)	access-use(r1)=1, def-use(r1)=1, useful-lifetime(r2)=1
I5	`r3 <= r1 + r2`	def(r3), use(r1), use(r2)	access-use(r1)=1, access-use(r2)=1, def-use(r1)=2, def-use(r2)=1

property of a read of a variable and each such read will generally have a different def-use interval associated with it. The exception occurs when two reads of the same variable occur in the same instruction (the same as was the case with the access-use interval above). Our definition for the def-use interval is also referred to as the variable instance *age* in the work by Franklin et al. [3] and Eeckhout et al. /citeeeckhout01hybrid. The idea of age is apparent since each use of a variable instance can be thought of occurring at a certain age of the instance as measured by the dynamic number of instructions since the associated definition of the same instance.

A simple, and quite contrived, code example to illustrate the meaning of the three intervals that we have defined is shown in Figure 1. In this simple code example, the term *c* is an arbitrary immediate constant encoded within the instruction which is also generally different for each instruction. All of the instructions produce defs of a variable instance associated with their destination registers. Instructions I3 and I4 also produce uses of the registers *r2* and *r1* respectively. The def of register *r1* in instruction I3 allows for the determination of the useful-lifetime for the previous variable instance (from instruction I1) held in that register. In the present example, the useful-lifetime for that variable instance is calculated as 0. This result is due to the fact there there were no intervening uses of the register between instructions I1 and I3. Likewise, register *r2* is defed in instruction I4 and this allows for the determination of the useful-lifetime for the previous variable instance held in that register (from I2). That is calculated as being being 1 since there was a use of that previous variable instance in instruction I3. Note that useful-lifetimes for a variable instance cannot be determined until a subsequent write of the corresponding variable is encountered. Similarly, both instructions I3 and I4 contribute a def-use interval with the value 1 since each represents a use of the corresponding variable instance just one instruction after its associated def. Finally from instruction I5, since there are uses of both registers *r1* and *r2*, an access-use and def-use interval can be determined for each of these. Note that for register *r1* an access-use interval with value 1 is determined while a def-use interval of value 2 is determined. For register *r2*, both its access-use and def-use intervals have value 1 since there was a def of that register in just the previous instruction.

Note that an access-use interval and a def-use interval is always determined for each use of a variable instance, but that a useful-lifetime interval can only be determined on a def of a new variable instance. Although this example illustrated the determination of the three types of access intervals on registers, these are determined similarly for memory references.

3.2 Microarchitectural significance of the intervals

Each type of variable access interval that we are exploring may have different significance or consequence for making distributed microarchitectural design decisions. Although all of the intervals discussed previously share some similarity in their attributes, they tend to answer different microarchitectural questions. Firstly some types of intervals are associated with reads while others are associated with writes. This distinction is used when considering the applicability of each type of interval metric.

The useful-lifetime interval is a property of a write so data on these intervals would be useful when exploring the desired microarchitectural consequences of a write occurring. If, for example, the resulting operand from a write operation could be buffered locally near the execution units in the processor, on the average it would only have to remain buffered until a number of following instructions (in dynamic program order) equal to its useful-lifetime had an opportunity to snoop the buffer for the operand. After the appropriate number of subsequent instructions had an opportunity to snoop the buffer, it could be assumed that the likelihood of a further use of that operand is minimal and therefore the operand could be released or evicted from the buffer.

In contrast, since both the access-use interval and the def-use interval is a property of a read, microarchitectural decisions about how to best satisfy read operands might use the data associated with either of these types of access intervals. If we first consider the case of no intervening buffering or caching of operands between an operand sourcing instruction and its associated sink instructions, then the def-use interval data would be appropriate to use for possible design decisions. This is so because operand sinking type instructions could only be satisfied by either a previous operand sourcing instruction (allowing for operand bypass) or lacking that, the operand would have to be fetched from the appropriate centralized resource (architected register or future file in the case of a register, and the L1 data cache or load-store queue in the case of a memory operand).

For those distributed microarchitectures that can employ some sort of operand buffering or caching spatially close to the execution units, the access-use interval is the more appropriate metric to determine design decisions. This is so because a cache of some sort can retain a copy of the desired operand even though the original generation (original write) of the operand may have occurred in the distant past in the instruction stream. Either the cache could maintain the operand for a reasonably long period of time from a preceding write, or perhaps more likely, intervening reads keep the cache from evicting the operand needed by subsequent sink instructions. Further, a previous read could have resulted in the operand getting cached from the appropriate centralized resource without even a recent write of the operand having ever occurred. An example of a distributed microarchitecture that uses interspersed caching of operands among the execution units is presented in Section 4.

4 Benchmark statistics

In this section we present the results of accumulating our access interval data on ten benchmark programs. The programs are taken from the SpecINT-95 and SpecINT-2000 suites. The following programs were used from SpecINT-95: GO, COMPRESS, IJPEG. The following programs were used from SpecINT-2000: BZIP2, CRAFTY, GCC, GZIP, MCF, PARSER, and VORTEX. All programs were primarily compiled for the MIPS-1 ISA, with a few MIPS-2 and MIPS-3 instructions included as a consequence of executing any system library code. The Silicon Graphics (SGI) vendor compiler was used for all compilations under the Irix 6.4 operating system. The programs were optimized using the '-O' compilation flag. All programs were executed for 600 million instructions but data was only collected and processed after skipping the first 100 million.

The next subsection presents the results for register access use. The following subsection presents the results for the memory access use. For both registers and memory, data is shown for access-use, def-last-use (useful lifetime), and def-use intervals.

4.1 Register access interval results

In this section, we show the register access interval results for our ten benchmark programs. For paper-space reasons we only present average interval percentage density and distributions. Figure 1 shows the three types of access intervals (top to bottom respectively): access-use, useful-lifetime, and def-use. For each type of access interval, the data for each of the ten benchmark programs are overlaid on the same graph. For each access interval type, both a density and a distribution is provided. Viewing the results across all benchmark programs, some general observations can be made. For the access-use intervals on average about 85% of all intervals are within 10 dynamic instructions and about 95% of all intervals are within 20% instructions (and about 97% within 25 instructions). For the useful-lifetime intervals about 84% of all intervals are within 10 instructions and about 92% are within 20 instructions. Finally for the def-use intervals, about 70% of them are within 10 instructions, about 80% within 20 instructions, and about 82% within 25 instructions.

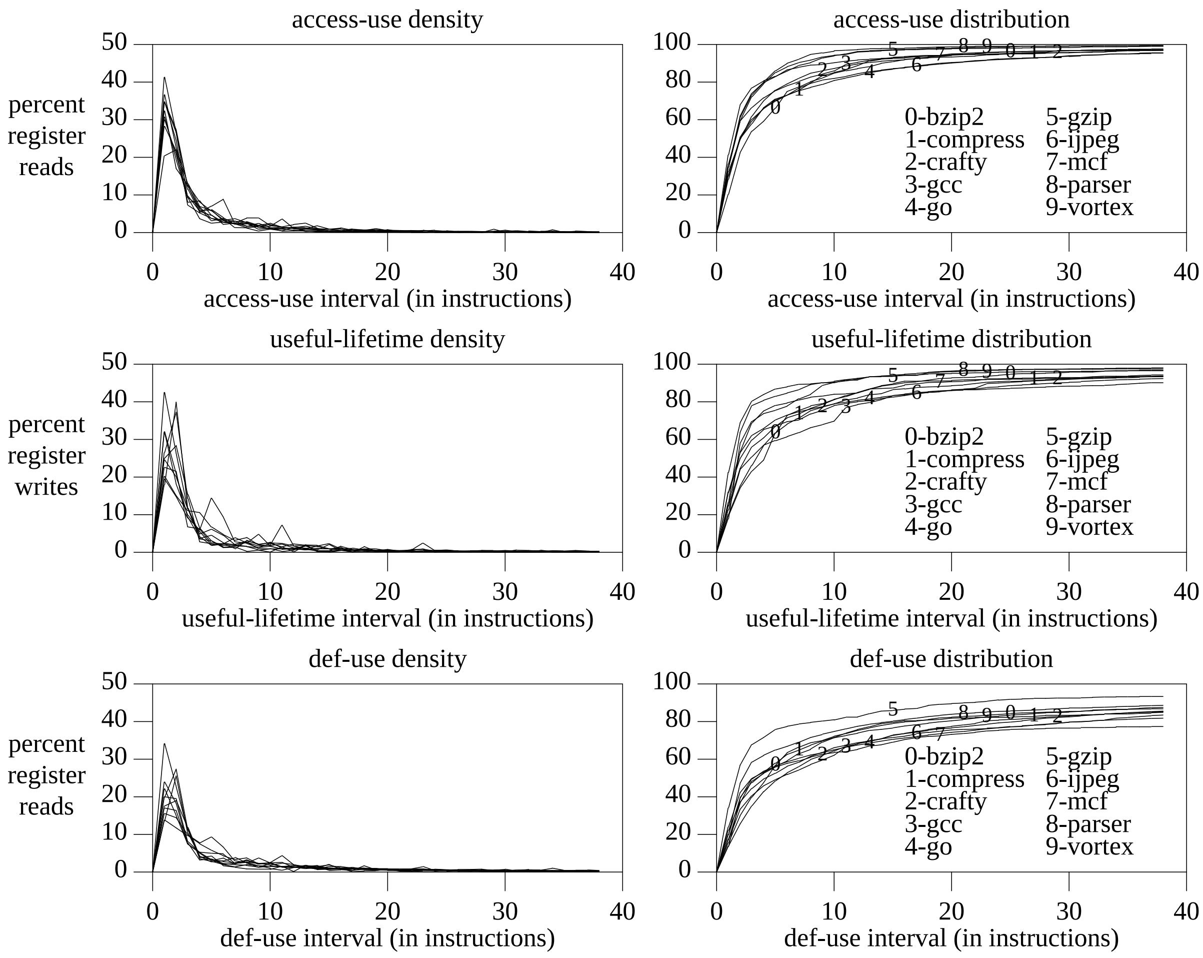

Figure 1: *Register Access Intervals.* Data results for all programs are shown overlaid. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions. For the distribution graphs, numbers corresponding to each benchmark program are sporadically associated with their percentage data lines in the graphs.

These results indicate that if, for example, operand storage units or caches are placed at intervals of approximately 10 to 20 instruction execution units apart that the better part of most of the register operand traffic would not even need to pass through a unit in order to be delivered to the next instruction needing that operand. For those operands that do need to pass through an operand storage unit, likely only one such unit needs to be traversed before delivering the operand to the instruction needing it. Since any operand storage unit incurs a delay in passing an operand through it, the smallest number of such units in the path of each operand is desirable.

4.2 Memory access interval results

In this section, we show the memory access interval results for the same set of benchmark programs. The data are presented in three sets. The first set of data is for register access-use intervals. The second set of data is for register useful-lifetimes, and the third is for register def-use intervals.

The data for memory access-use intervals are shown in Figures 2 and 3. Results from benchmark programs BZIP2, COMPRESS, CRAFTY, GCC, and GO are shown in Figure 2 while the results from programs GZIP, IJPEG, MCF, PARSER, and VORTEX are shown in Figure 3. The data for memory def-last-use (or useful-lifetime) intervals are shown in Figures 4 and 5. The data for register def-use intervals are shown in Figures 6 and 7. In Figure 8 we show the cumulative data over all benchmark programs. That figure shows all three of the access intervals that we explored: access-use, useful-lifetime, and def-use. As can be seen from the various figures for the memory access intervals, unlike for the register intervals, these are significantly varied from one benchmark program to another. The more prominent spikes in several of the density graphs is evidence of high frequency looping over memory. It is interesting to note that GCC (and to a lesser extent GO) has some pronounced looping behavior for its access-use intervals but not for its useful-lifetimes nor def-use intervals. This represents more repeated reading of constant memory variables rather than the more varied read-write memory behavior of most of the other programs. Also, it is interesting to note that most programs exhibit a large number useful-lifetimes of 0. A zero-length useful lifetime represents a memory location that is later overwritten without an intervening read (an abandoned write). This can happen when writing to output file buffers, for example, but can also be due to control flow changes that abandon written variables. The zero-length useful-lifetimes vary from about 11% and 12% of all program writes for programs such as BZIP2 and GZIP respectively, to 20% to 61% with GCC, CRAFTY, and VORTEX having the highest number of abandoned writes as compared with the others.

5 Summary

We have presented data for various access intervals associated with both register and memory variable instances. The data for the register intervals is consistent with and confirms the prior work by Franklin et al. That data shows that for most general purpose sequential program codes (for example SpecINT) as many as 82% of the register reads (a use of the variable instance) will have had their instances defined within the preceding 25 dynamic instructions (this is drawn from the def-use results). This indicates that for large instruction window sizes (where 25 is a small fraction of the size) that there is a good chance that the associated register operand can come directly from the defining instruction without having to be first stored in either the architected register file or some other centralized resource (such as a future file). When some form of register buffering or caching is employed, it can be expected that the percent of register reads satisfied within the execution window will be even higher, and could average about 97% of all reads having been either defined or present in an operand cache located within the last 25 dynamic instructions (from the access-use interval data).

For memory variable instances, there is not as much likelihood of any given memory read (*load instruction*) having its variable instance already within the execution window (as is the case with register variable instances). However, for memory reads, the cost of having to go outside the execution window of the processor is higher than as for registers and any possible operand bypass of either the load-store queue or the L1 data cache is still welcomed. Our data shows that about 30% of memory read operations can expect to have their preceding instance definition within the last 100 dynamic instructions (def-use characterization results). Although this is not as useful for operand bypass, as in the case of registers, any bypass that can occur in these cases reduces the access burden on both the load-store queue as well as the L1 data cache. Similarly to the situation for registers, greater percentages of reads (memory load requests) can be expected to be satisfied within the execution window alone (bypassing the LSQ and L1 cache) when distributed caches (distributed L0 caches) are employed. From the access-use interval characterization data, about 55% of all memory loads could benefit by operand bypass of centralized resources for machines that can have the same previous 100 dynamic instructions within the window. Although more memory reads can be satisfied with larger machine execution windows, the returns are diminishing.

Since future large microarchitectures (defined as having large or very large numbers of execution units) will likely be forced to be physically (spatially) distributed across integrated circuit or multichip modules, the use of small operand storage or caching units (spatially placed among the execution units) to facilitate operand bypass of the architected register file or the L1 data cache may be essential. The access interval data presented, for both registers and memory, will be very useful for guiding design decisions about the employment of these operand storage or caching units.

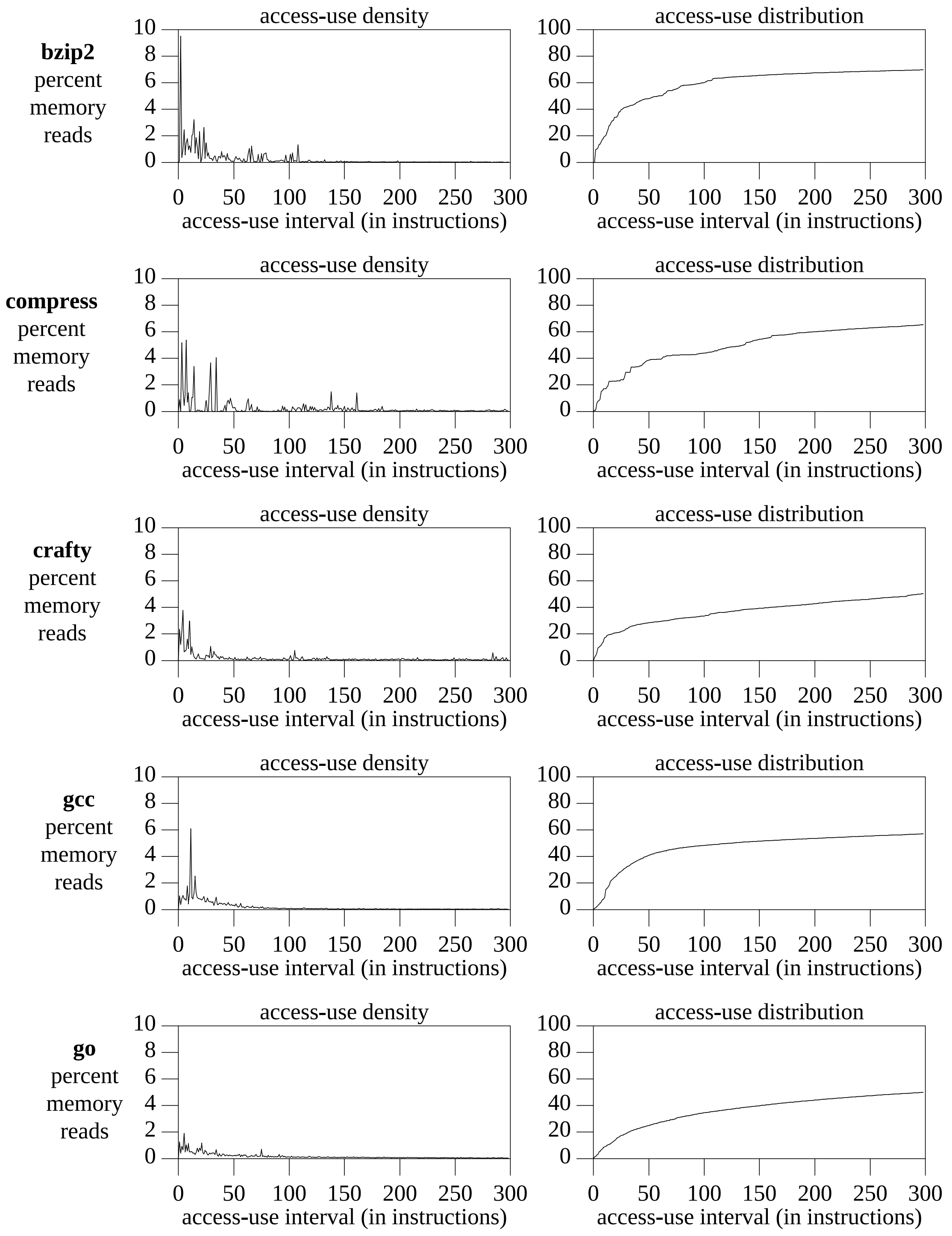

Figure 2: *Memory Access-Use Intervals.* Data results for the BZIP2, COMPRESS, CRAFTY, GCC, and GO programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

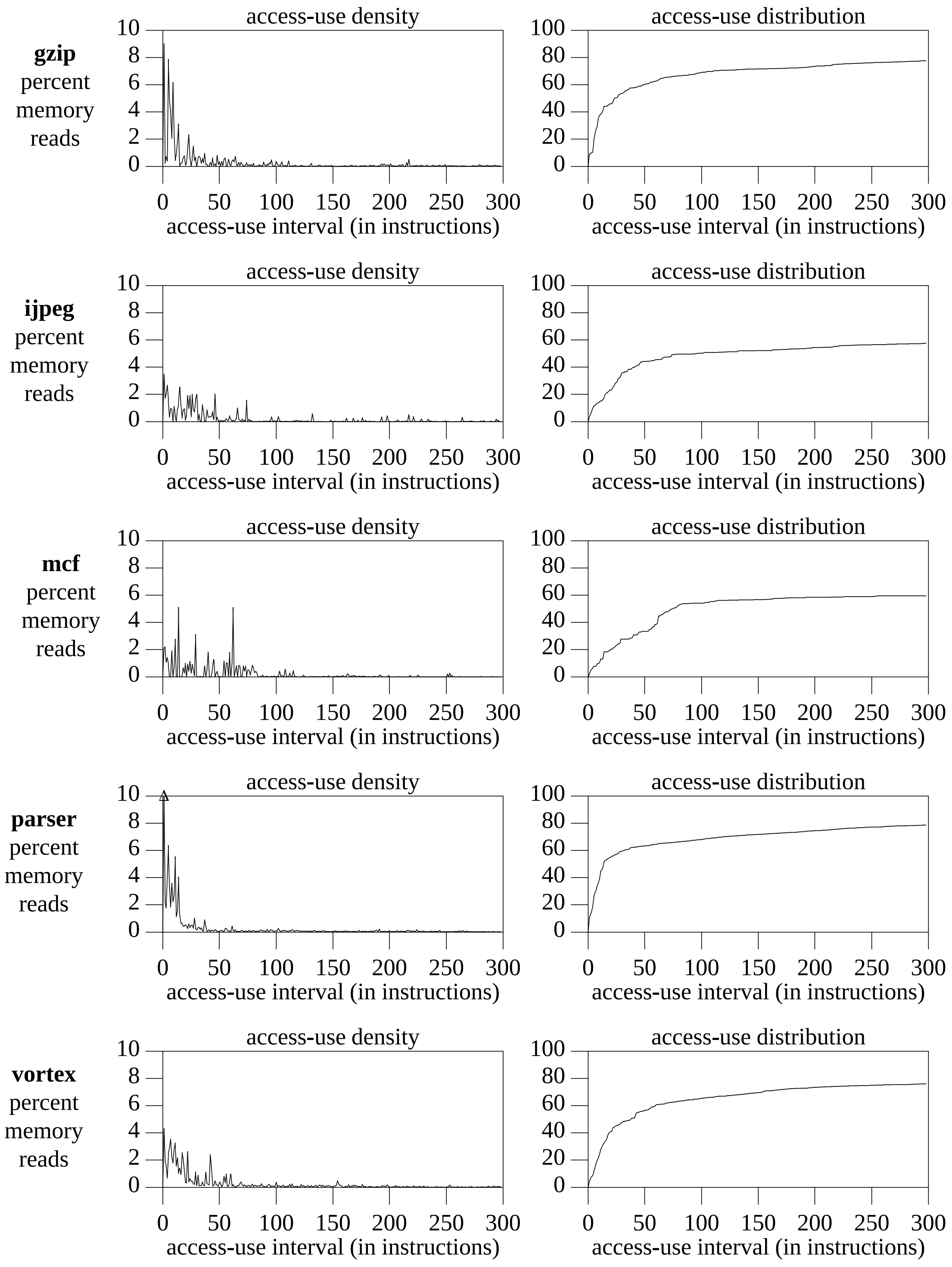

Figure 3: *Memory Access-Use Intervals.* Data results for the GZIP, IJPEG, MCF, PARSER, and VORTEX programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

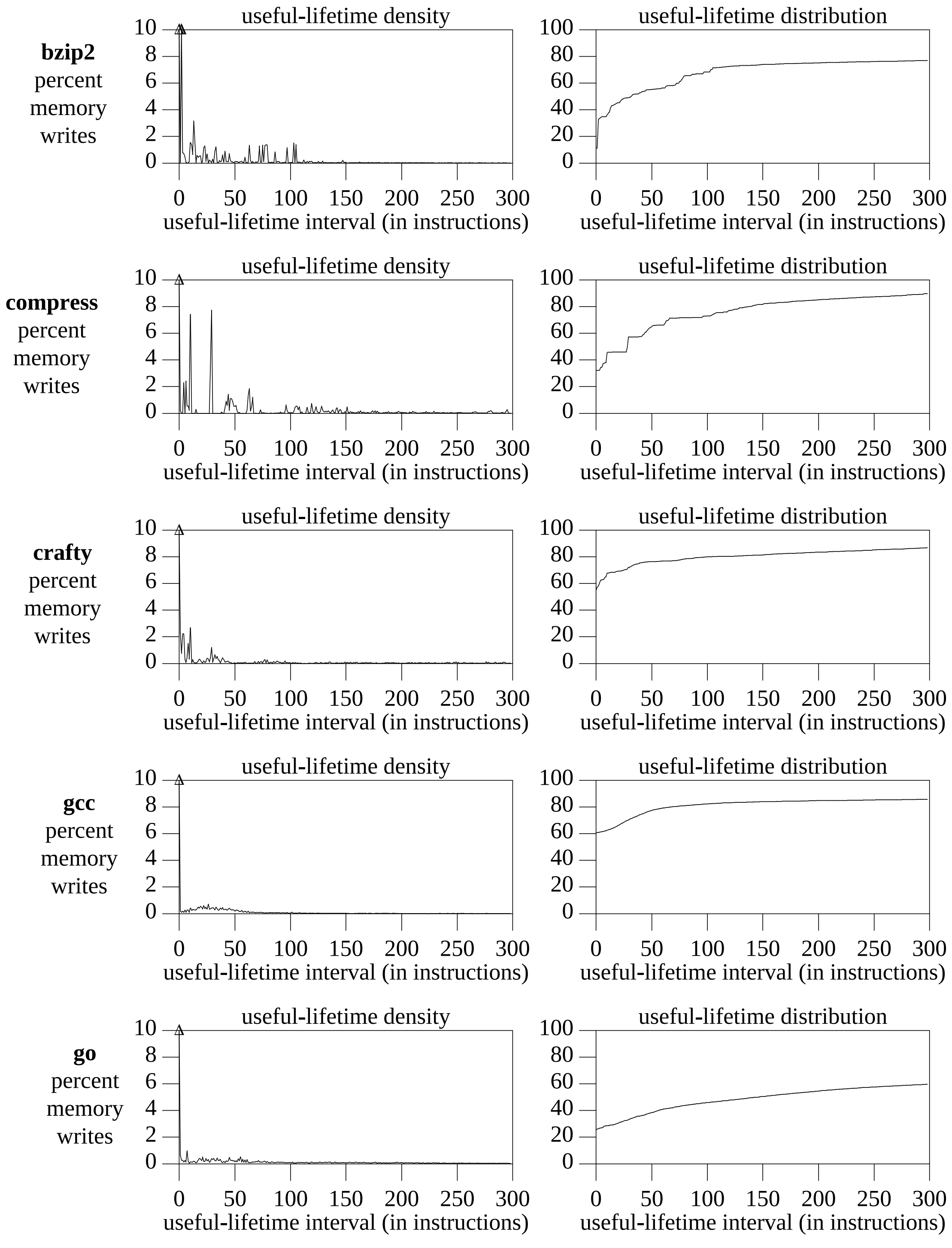

Figure 4: *Memory Def-Last-Use Intervals.* Data results for the BZIP2, COMPRESS, CRAFTY, GCC, and GO programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

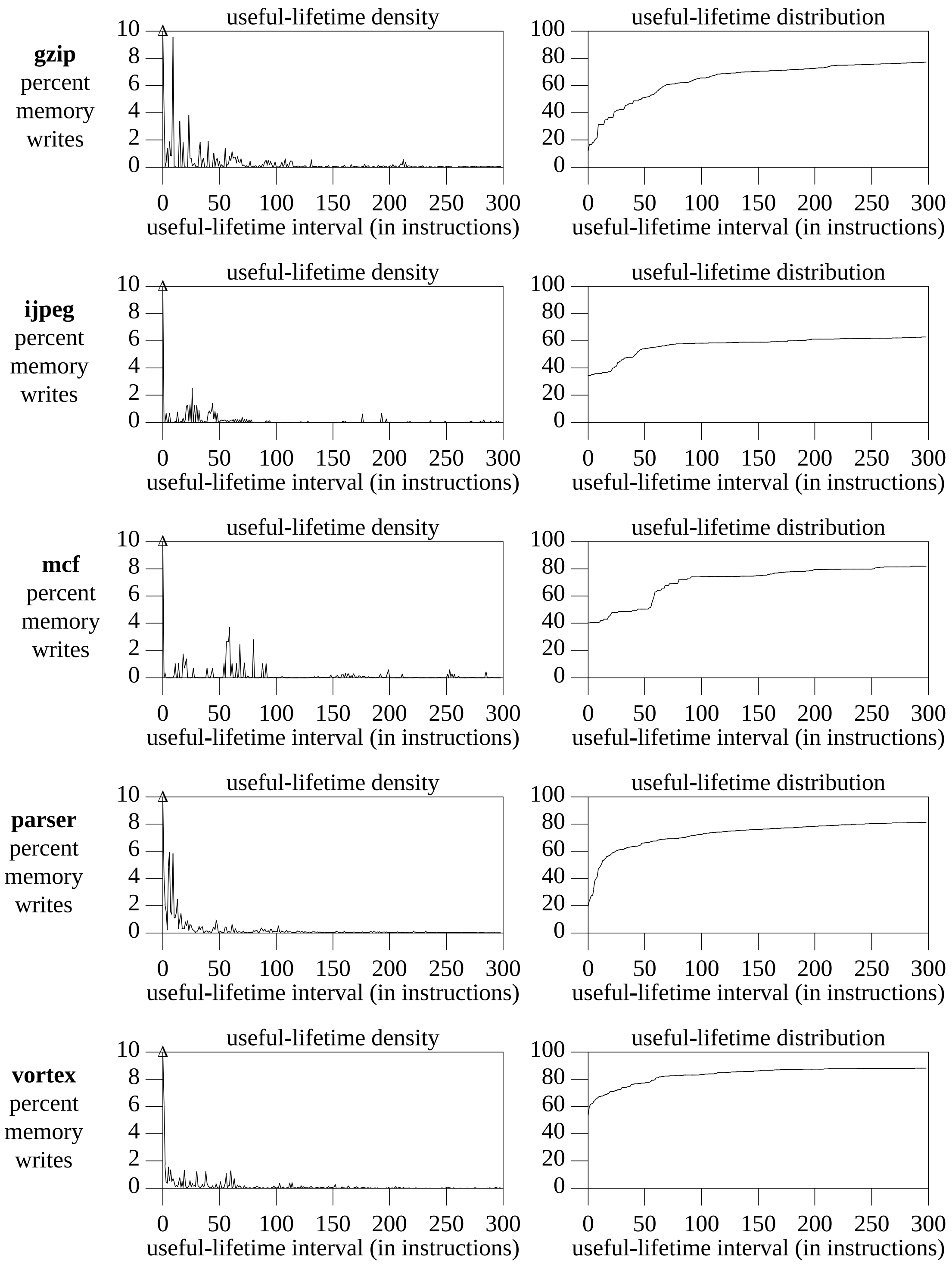

Figure 5: *Memory Def-Last-Use Intervals.* Data results for the GZIP, IJPEG, MCF, PARSER, and VORTEX programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

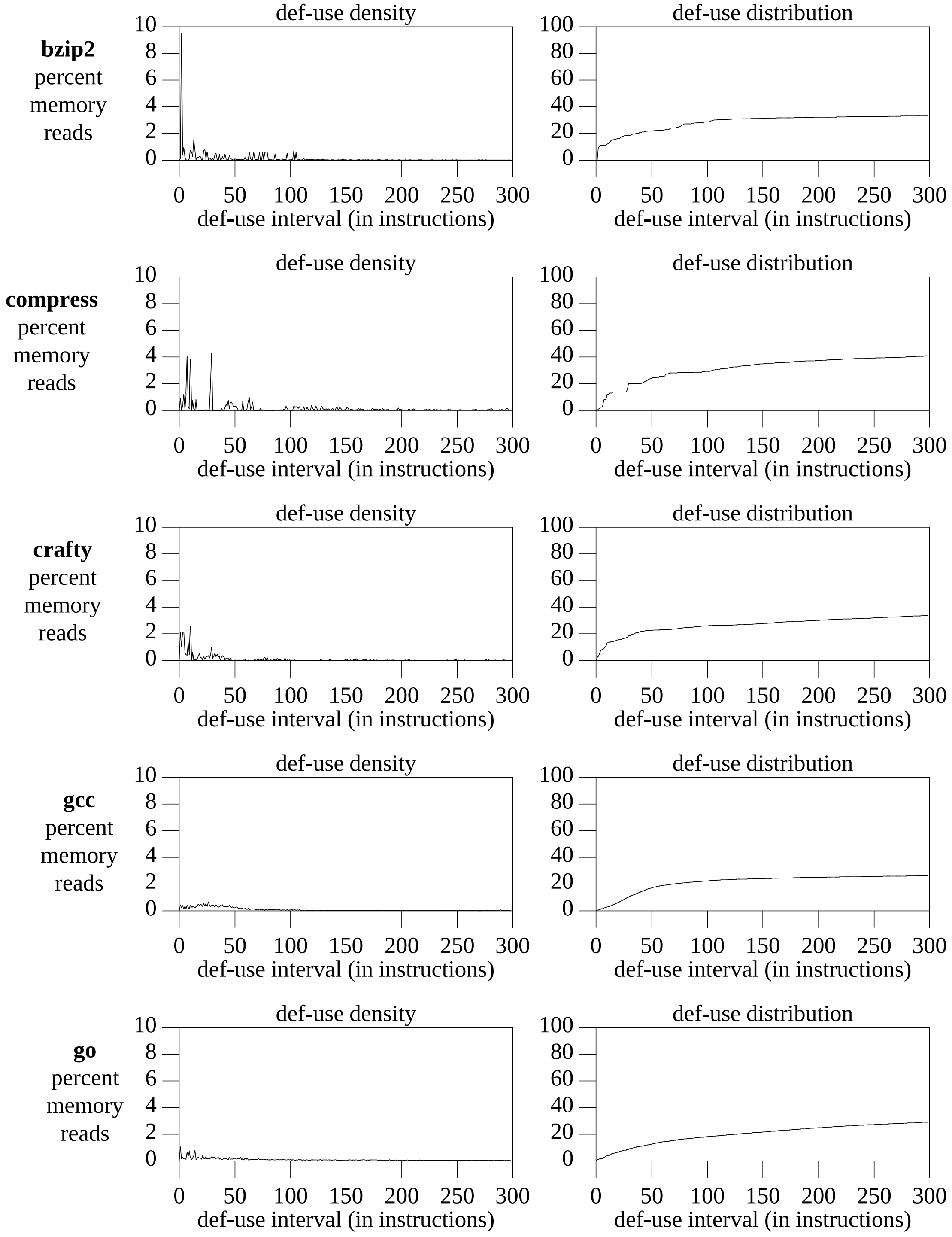

Figure 6: *Memory Def-Use Intervals.* Data results for the BZIP2, COMPRESS, CRAFTY, GCC, and GO programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

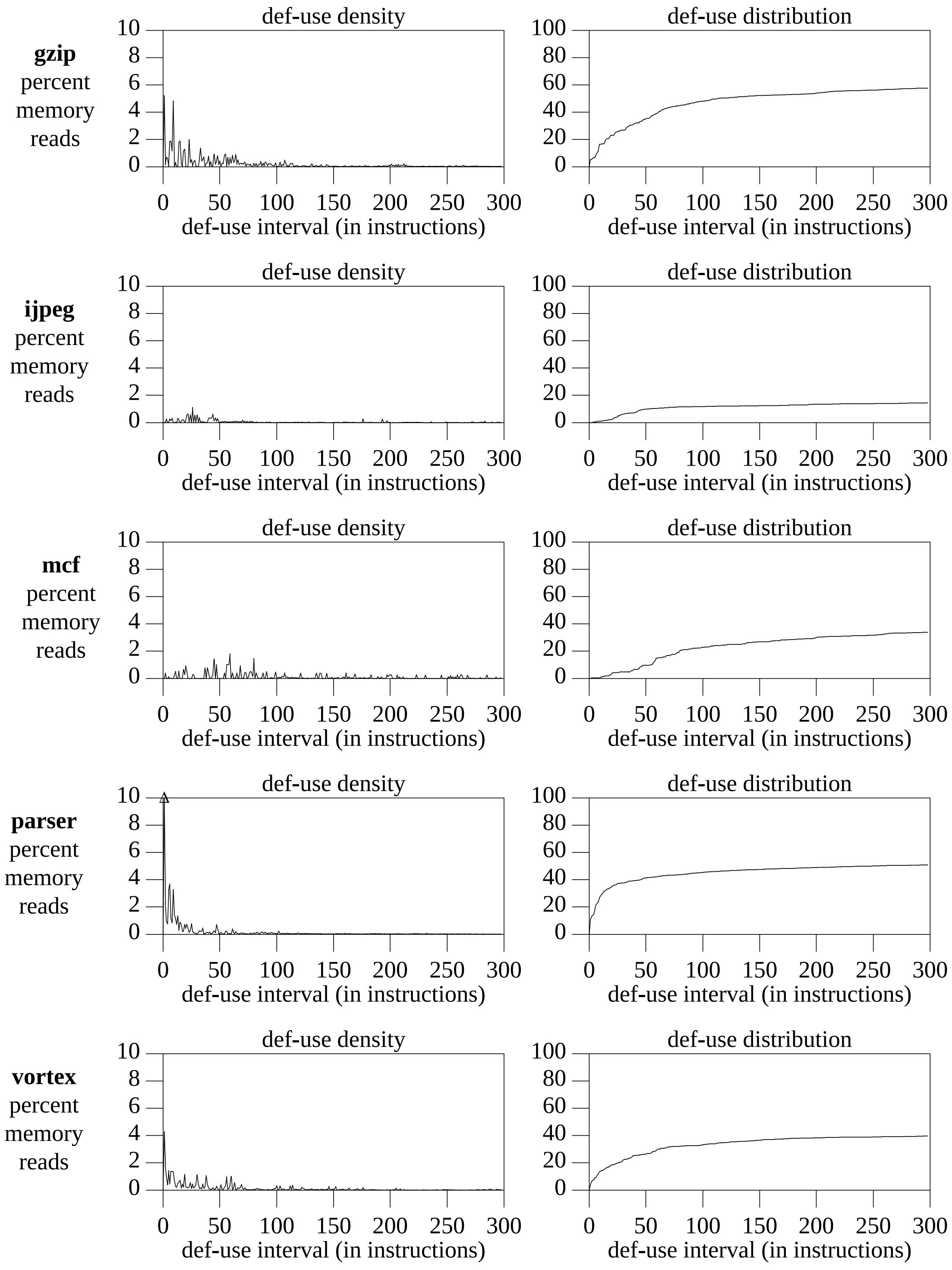

Figure 7: *Memory Def-Use Intervals.* Data results for the GZIP, IJPEG, MCF, PARSER, and VORTEX programs are shown. The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

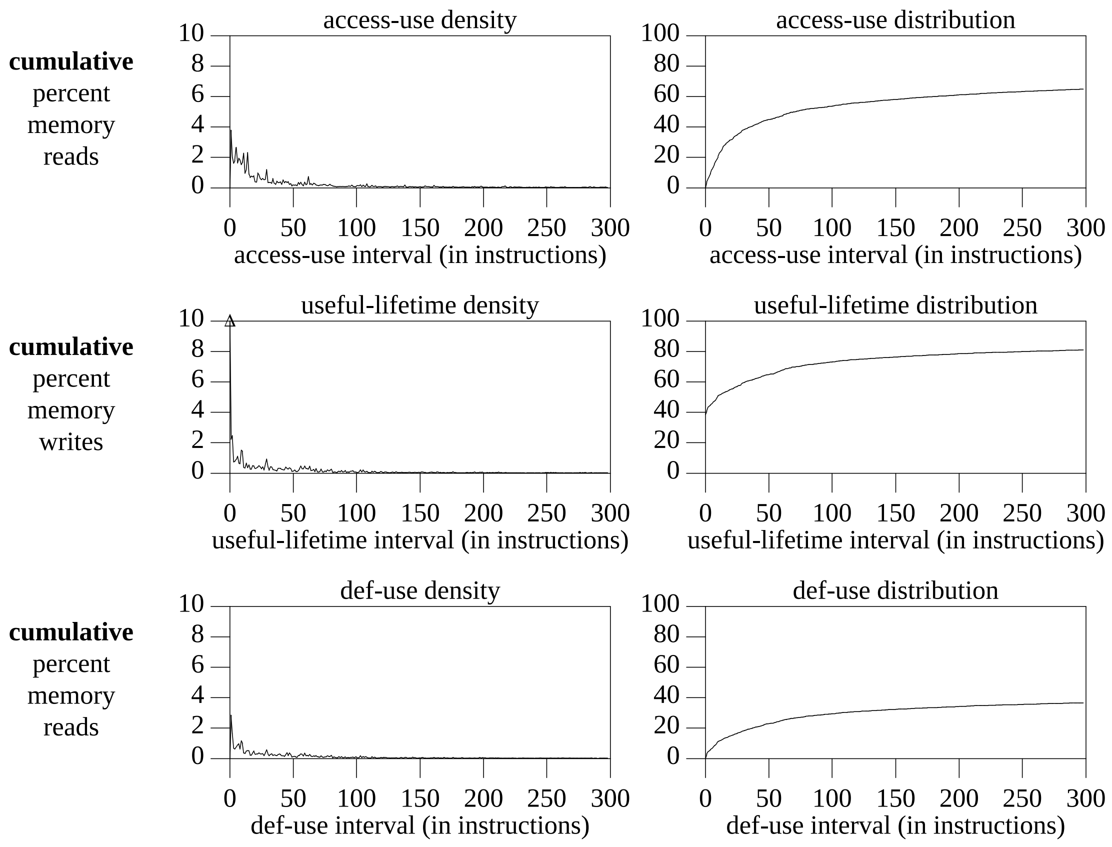

Figure 8: *Cumulative Memory Intervals over all benchmarks.* The density is shown on the left and the distribution is shown on the right. All intervals are measured in dynamic numbers of executed instructions.

References

[1] Eeckhout L. and Bosschere K.D. Hybrid analytical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, September 2001. IEEE Press.

[2] Farkas K.I., Chow P., Jouppi N.P., Vranesic Z. The multicluster architecture: Reducing cycle time through partiioning. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 149–159, 1997.

[3] Franklin M. and Sohi G.S. Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grained Parallel Processors. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 236–245, New York, NY, Dec 1992. ACM Press.

[4] Hennessy J.L. and Patterson D.A. *Computer Architecture: A Quantitative Approach, 2nd ed.* Morgan Kaufmann, Palo Alto, CA, 1995.

[5] Henry D.S., Kuszmaul B.C., Viswanath V. The ultrascalar processor – An asymptotically scalable superscalar microarchitecture. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI.* IEEE, Mar 1999.

[6] Jiser J., Carr S. and Sweany P. Global register partitioning. In *Proceedings of the 9th International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, October 2000. IEEE Press.

[7] Kemp G.A., Franklin M. PEWs: A decentralized dynamic scheduler for ILP processing. In *Proceedings of the 24th International Conference on Parallel Computing*, pages 239–246, 1996.

[8] Madison A., Bunt R. Characteristics of program localities. *Communications of the ACM*, May 1976.

[9] Morano D.A., Khalafi A., Kaeli D.R., Uht A.K. Realizing high IPC through a scalable memory-latency tolerant multipath microarchitecture. In *Proceedings of MEDEA Workshop (held in conjuction with PACT'02)*, 2002.

[10] Nagarajan R., Sankaralingam K., Burger D. and Keckler S.W. A design space evaluation of grid processor architectures. In *Proceedings of the 34th International Symposium on Microarchitecture*, New York, NY, Nov 2001. ACM Press.

[11] Rotenberg E., Jacobsom Q., Sazeides Y. Trace processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 138–148, 1997.

[12] Rotenberg E., Smith J. Control independence in trace processors. In *Proceedings of the 32th International Symposium on Microarchitecture*, 1999.

[13] Sohi G.S., Breach S. and Vijaykumar T.N. Multiscalar Processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, New York, NY, Jun 1995. ACM Press.

[14] Sundararaman K.K., Franklin M. Multiscalar execution along a single flow of control. In *Proceedings of the International Conference on Parallel Computing*, pages 106–113, 1997.

[15] Tomasulo R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

[16] Tsai J-Y., Yew P-C. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, 1996.

[17] Uht A.K., Morano D.A., Khalafi A., de Alba M., Kaeli D. Realizing high IPC using time-tagged resource-flow computing. In *Proceedings of the the EUROPAR Conference*, Aug 2002.

[18] Vajapeyam S., Mitra T. Improving superscalar intruction dispatch and issue by exploiting dyanmic code sequences. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[19] Verkamo A. Emperical results on locality in database referencing. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Aug 1985.

[20] Vidyadhar P., Bhaskarpillia G. An inter-reference gap model for temporal locaility in program behavior. In *Proceedings of SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 191–300. ACM Press, 1995.