ICOT Visit Report

Evan Tick
Department of Computer Science
University of Oregon
Eugene, OR 97403 USA

June 6–20 1992

1 Introduction

This report summarizes my two-week visit to ICOT, working primarily in the First Research Laboratory under the direction of Dr. Taki. The visit was an extension of my participation in the FGCS'92 conference and Advanced Architecture workshop of the previous week. The main goal of this visit was to continue research with A. Imai on efficient parallel garbage collection methods for concurrent logic programs. In this paper I will summarize our results of these two weeks of collaboration. Furthermore, my visit allowed me to have fruitful discussions with K. Ueda, M. Morita, T. Chikayama and K. Kahn (also visiting ICOT) on topics of mode analysis, compilation techniques and code generation, and visualization.

2 Overview of Schedule

My schedule was rather tight, restricted further by a visit to Tsukuba University for two weekdays. My general schedule was as follows:

Mon Demonstrated the VISTA visualization tool to Imai-san and began discussions on our GC research strategy.

Tues Demonstrated VISTA to M. Sato and other Oki members. Had extensive discussions about Janus, moded FGHC, and visualization with Ueda-san, Morita-san, and K. Kahn.

Wed Reviewed GC papers concerning concurrent logic programs, and wrote literature review of Nakajima's "piling" GC, Ozawa's two-generation GC, and Koike's distributed two-generation global GC. Had discussion with Nakajima-san clarifying the "piling" method, and our own ideas.

Thurs Gave PIM-WG lecture this afternoon on our Monaco system. Worried about poor Monaco performance in evening.

Fri Discussed our mode analysis tool with Morita-san. Discussed KL1-into-C compiler with Dr. Chikayama.

Mon Visit this afternoon to Dr. Koike at Tokyo University, where he demonstrated PIE64, and the various FLENG implementations. We had an extensive discussion about his distributed GC scheme, and our ideas.

Tues Discussion with Aiba-san about "top-down technology" *vs.* "bottom-up technology," and the future of constraint language technology and GDCC in particular. Wrote a second draft of the "ICOT Evaluation Report."

Wed Tsukuba University lecture series on "Concurrent Logic Programming."

Thurs Tsukuba University lecture series. Returned to ICOT in evening to continue GC work.

Fri Wrote "ICOT Trip Report." Continued planning GC work.

3 Garbage Collection Research

In this section I will summarize the direction of joint research with A. Imai in developing an efficient parallel garbage collection algorithm. Previous work [3] concerned a parallel stop-and-copy algorithm for operation on a shared-memory multiprocessor. Since the implementation of that scheme on PIM, it was determined that certain objects, such as code modules, create a lot of useless work because they are copied back and forth during successive garbage collections. The standard method of solving this problem is the incorporation of *generation scavenging* [11].

Our main interest is to exploit generation-type garbage collection in conjunction with our parallel semi-space algorithm that automatically balances the load. In the context of generation-type GC, two main problems are incurred by concurrent logic programs. Logical variables can be bound to point from an old generation into the current work space. When garbage collecting the current space, no root pointer will reach these live cells. A second problem is similar: incremental garbage collection, such as MRB [2], allows cells to be reused. Thus a cell in an old generation may be overwritten with a pointer into the current space.

Our solution to these problems is to disallow terms with unbound variables (anywhere within the term) from being interned in an old generation. Furthermore, although terms with MRB-off components may be interned, the MRB reuse mechanism will be modified to avoid reusing such cells if they appear in an old generation. Another alternative is to intern only MRB-on structures.

To obviate costly checks for points from an old generation into the current generation, no such pointer should ever be allowed to be created. Thus we cannot intern a non-fully-ground term. With respect to MRB, however, we have some flexibility. We could outlaw interning a term with some MRB-off subcomponent, or we could allow it, but disallow reusing the individual MRB-off cell. The latter solution is cheaper to implement, and causes no inefficiency if the cells that are initially MRB-off evolve to MRB-on, in which case they cannot be reused anyway.

However, it requires some additional runtime overhead within the KL1-B "check" instructions which decide to reuse a cell or not. Given a strategy of trading off as much runtime overhead for GC overhead as possible, we choose the solution of restricting internment.

We are considering two methods of determining the groundness criterion for internment. The first alternative is to ensure fully-groundness at GC time only. This would entail traversing each term before copying during GC. The traversal, being a sequence of reads, would be execution overhead. However, cache performance will probably not suffer for average-size terms because the initial traversal will pull the term into cache, and the subsequent writes during copying will hit. Furthermore, MRB-on-ness can be checked during this traversal, for little additional cost. Although this method has some overhead, it is exact in the sense that every fully-ground active term will be interned. This is in contrast to the tag scheme described next, wherein we make conservative approximations to "fully ground" and thus we may miss some opportunity for internment.

An alternative scheme requires the definition of two types of terms: fully ground terms and not-known to be fully-ground terms. We are in general concerned with vectors (lists are a degenerate type of vectors that we will discuss later). Vectors have two distinct tags in our scheme: fully ground (FGV) or not known to be fully ground (NFGV). Initially, vector creation by initialization to a fully ground vector is tagged as FGV. For example, `X = {a,{b,c}}`. At runtime, whenever we update a vector element (with a builtin predicate for this purpose), the vector tag, and groundness of the new element, are checked. In the case that the vector is FGV and the new element is fully ground, the tag remains FGV. Otherwise the tag becomes NFGV. Note that the groundness of the new element is frequently known at compile time, thereby simplifying the check. However, the scheme does *not* allow a vector to move from NFGV to FGV, which would require more complex management.

Lists are special terms with two arguments. We do not need to allocate two tags to lists (FGL and NFGL) because of their simplicity. In most cases we can determine at compile time the groundness of a list pair, and if not, then the runtime check is not costly. A disadvantage to this scheme in general is that checking for MRB-on-ness would require a different mechanism.

All these schemes will have utility if the checking overhead is low, and the occurrence of internable terms is high. We believe fully ground structures are common in certain applications. In concurrent logic programs, logical variables are not "first-class citizens," as in logic programs. In other words, variables are overloaded to implement process synchronization. If a variable is unbound at the point of a particular procedure invocation, it is not certain if the value will be produced by concurrently executing processes. Thus the `var(X)` guard predicate loses its credibility in such a language. Programmers avoid this problem by *simulating* logical variables as in functional languages — with special atoms. These atoms are overwritten, representing binding. Thus manipulation of structures in this manner will result in fully ground terms that can be interned, avoiding extra copies during garbage collection.

Note that streams, through which most communication is performed in concurrent logic programs, are incomplete data structures and are never fully ground. Thus streams (usually

implemented with lists, but general terms can also be used) are never interned. Individual messages on streams will be interned if they are fully ground. Thus incomplete messages will not be interned. Our hypothesis is that streams have the shortest life of all terms, and incomplete messages don't live much longer. In general, communication infrastructure is built up simply to transmit information, and then is reused. Thus our scheme properly keeps these terms in the current generation.

Dr. Koike claimed that experience with FLENG garbage collection indicated that goal records can be both properly and improperly interned in an old generation. In most applications, suspended goals should *not* be interned because they will soon be resumed. However, fine-grain object-oriented applications can create many suspended goals that rarely get resumed. These deserve to be interned. Our current method, as stated, will automatically avoid internment of suspended goals, which by definition are waiting on an unbound input variable (usually a stream), and therefore are not fully ground.

Multiple generations can perhaps solve the long-life goal record problem. For example, a special generation for goal records, from which a tuned internment threshold filters perpetual tasks from standard tasks. However, any such scheme will necessitate runtime trail checking for *every* binding made in the machine.

3.1 Literature Review

We now briefly review the body of work in generation-type garbage collection with respect to logic programs, specifically committed-choice programs. Committed-choice programs do not backtrack as does Prolog, so that garbage collection is invoked more frequently. However, during collection, the view is almost the same. All this work is based on seminal work by Baker [1], Lieberman and Hewitt [5], and Unger [11].

Nakajima [8] first proposed a scheme for "piling" multiple generations on a shared-memory multiprocessor. Ozawa *et al.* [9] designed and evaluated a parallel two-generation collector on shared-memory multiprocessors. Koike and Tanaka [4] proposed a two-generation collector for distributed-memory systems. In all the schemes, the fundamental goal is the same: avoid collecting garbage within older generations, under the assumption that objects in these older generations have long lives.

3.2 Quantitative Garbology

"Garbology" is the environmental study of the composition and characteristics of garbage. We extended VPIM, a parallel KL1 emulator, for measuring a large set of medium-sized benchmark programs. The preliminary measurements presented here were collected on a Sequent Symmetry with 16 processors. We previously used VPIM to develop our parallel copying garbage collection algorithm [3]. Here we extended VPIM to simulate generation scavenging, first to confirm results by Ozawa *et al.* and to further distinguish the characteristics of memory objects, e.g., what is the correlation between lifetime and groundness?

Figure 1 shows the lifetimes of objects in some typical benchmark programs. These graphs are modeled after the data presented by Ozawa, for ease of comparison. A slight difference is that we force GCs every k^{th} reduction, and plot the active cells remaining in a generation *as a percentage of the active cells surviving the first GC.* Each plot the figures corresponds to a particular generation. Thus we see that objects within successive generations die off in distinctive patterns: rapid decrease followed by steady tails. As Ozawa pointed out, this data indicates that two-generation GC, will work well for committed-choice programs. More complex analysis is necessary to determine the optimal internment threshold, which strikes a balance between reduced copying and increased old generation compaction.

We simulated "virtual" two-generation GC by accounting for how many active cells we would avoid copying with respect to single-generation GC. This was implemented on VPIM to give us quick confirmation of Ozawa's results. Figure 2 shows the percentage savings (of copying) achieved by interning in the old generation space, for some typical benchmarks. Copying GC is the baseline of 100% for all heap sizes. As the heap grows, the savings decreases, as expected. Furthermore, as the threshold increases (the number of current generation GCs survived before internment), savings decreases.

Figure 3 shows two views of the garbage collection work per object in the Mastermind benchmark. Both graphs are distributions of words copied vs. object size. The first graph shows the reduction in words copied afforded by the two-generation scheme. The second graph shows the actual words copied with the two-generation scheme. In other words, referring to Figure 2, these two distributions correspond to 80% and 20% of the work normally entailed in copying GC, respectively. The most important characteristic of the distribution is the peak at objects of size 1000 words. This increases the mean, increasing the utility of the scheme.

3.3 Future Work

I will now summarize the work that remains to be done in this project.

- Show correlation between fully-groundness and long life of terms.
- Analyze characteristics of MRB-on/off in the old generation, to determine the opportunity cost of avoiding trailing.
- If previous two results are promising, the next step is to fully specify and implement the hybrid algorithm on VPIM.
- Confirm that two-generation scheme removes our current problem with code object copying. Measure total reduction in work afforded by the scheme.
- Experiment with thresholds to determine an optimal strategy.
- Compare the tag *vs.* naive scheme for determining fully-ground terms. If previous measurements show that fully-ground constraint is too restricting, then we should implement a trail.

4 Compilation Research

Discussions with Chikayama-san, Ueda-san, Hirata-san and others, concerning compilation techniques for committed-choice languages, educated me about several compilation issues I previously did not understand. The relationship between FGHC, moded FGHC, kernel Janus, and Janus was clarified. The main distinction, as I see it now, is that moded FGHC allows multiple consumers of a stream, and also requires the compiler to determine stream producers and consumers. Kernel Janus allows streams with only single producer/consumers pairs, called the "teller" and "asker" of a stream, respectively. The programmer is required to specify this. However, Janus, built on top of kernel Janus, can allow multiple consumers. There are two methods for implementing this. Separate copies can be made of the stream to be broadcast, thereby retaining the single producer/single consumer constraint, and allowing memory reuse optimizations. Otherwise, without copying, memory reuse optimizations must be bypassed.

Similarly, moded FGHC implementations have the option of implementing multiple consumers by copying. This would always allow memory reuse optimizations on channels, with the system being responsible for determining the channels.

Comparing our Monaco compiler with the KL1-into-C compiler underway by Chikayama, I realized that several beneficial optimizations techniques are needed to achieve high performance:

- Static type analysis, groundness analysis, and hook analysis are essential. This was further emphasized by discussions with Koike, concerning GFA, the Global Fleng Analyzer. Although we have an analysis algorithm for Monaco mapped out, developed by a student at the University of Oregon, it has not yet implemented.

- Register allocation afforded by compilation into C is also essential. Most of Monaco's performance problems stem from extremely poor register allocation due to the final assembly stage from intermediate code into 80386 code. In fact, neither Chikayama or Koike even have register allocators in their compilers!

- For that matter, they don't have data flow analysis, such as common subexpression elimination and dead code elimination. This is not necessary because the final compilation from C does this.

- Unification should have at least one special case that is generated in-line: the case of one unbound variable term. Both Chikayama and Koike have this inline, with all other cases dropping off into function calls. Currently, Monaco does not have any inline check.

- Related to the previous point, Ueda suggested a method of renaming an assigned variable within a recursive procedure, so that it could be guaranteed that the variable is not hooked at the time of assignment. The renamed variable can later be unified to the original passed variable after the recursion has terminated (how this order is guaranteed is not clear). The advantage is removal of hook checks during assignment. Koike removes such checks by static analysis. Note that Chikayama has no such optimizations in his compiler.

- Koike shows in the Luna implementation of XFleng that compilation into C can still be made to execute in parallel. We need to study the structure of his system more carefully.

All these considerations has made me quite eager to start work on a moded FGHC into C compiler based on our Monaco type analyzer, our mode analyzer linked to a thread sequentializer [7, 6], and exploiting decision graph code generation as we currently do in Monaco. I believe this will give us a structural edge over the current Fleng and KL1 compilers, which to my knowledge do not use sophisticated indexing code generation or sequentialization. My initial goal in such an endeavor would be to show the utility of sequentializing threads. However, the general framework is good for a high-performance compiler. I am still worried about memory reuse, wondering if our current method [10] is powerful enough.

5 Thanks

I sincerely thank Director Fuchi, Dr. Uchida, and Dr. Taki for supporting my visit to ICOT. This was an especially busy time for ICOT researchers considering the effort that went into executing FGCS'92, and I appreciate the welcome they offered me.

I believe my work with A. Imai has progressed nicely. We began thinking about generation scavenging in December 1991 during Imai-san's visit to the University of Oregon. These initial discussions have now progressed into preliminary research results that are encouraging. The VPIM simulator was a key factor in allowing us to complete as much as we did in the past two weeks — we essentially confirmed all of the Fujitsu GC results [9] in one week! We hope that we can quickly check the utility of restricting internment, leading to our third research phase of building a fully parallel prototype by the end of August.

I would also like to thank Kumon-san (and all the ICOT researchers involved in various research discussions) for the valuable help and instruction they provided during my visit.

A good friend, Iwata-san, organized my visit, and generously invited me to dinner at his home. Thanks also go to Iwata-san's wife, for an excellent meal (as always), and providing the opportunity for a relaxed discussion with Y. T. Chien that night. During the FGCS conference and afterwards, Ms. Karakawa and Ms. Higuchi helped me in many matters, and I genuinely thank them for their friendship. A final word of thanks goes to the First Laboratory's Monthly Party Group, ablely led by Hirata-san. Even for a veteran like me, *that* night was memorable.

References

[1] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[2] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *International Conference on Logic Programming*, pages 276–293. University of Melbourne, MIT Press, May 1987.

[3] A. Imai and E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *IEEE Transactions on Parallel and Distributed Computing.* in press.

[4] H. Koike and H. Tanaka. Generation Scavenging GC on Distributed-Memory Parallel Computers. In *Proceedings of High Performance and Parallel Computing in Lisp.* EUROPAL Workshop, London, November 1990.

[5] H. Lieberman and C. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, 1983.

[6] B. C. Massey. Sequentialization of Parallel Logic Programs with Mode Analysis. Master's thesis, University of Oregon, September 1992. Also available as Technical report CIS-TR-92-18.

[7] B. C. Massey and E. Tick. Automatic Mode Analysis for Concurrent Logic Programs: Implementation and Evaluation. Technical Report CIS-TR-92-09, University of Oregon, Department of Computer Science, November 1992.

[8] K. Nakajima. Piling GC: Efficient Garbage Collection for AI Languages. In *IFIP Working Conference on Parallel Processing*, pages 201–204. Pisa, North Holland, May 1988.

[9] T. Ozawa *et al.* Generation Type Garbage Collection for Parallel Logic Languages. In *North American Conference on Logic Programming*, pages 291–305. Austin, MIT Press, October 1990.

[10] R. Sundararajan, A. V. S. Sastry, and E. Tick. Variable Threadedness Analysis for Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 493–508. Washington D.C., MIT Press, November 1992.

[11] D. Unger. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. ACM Press, 1984.

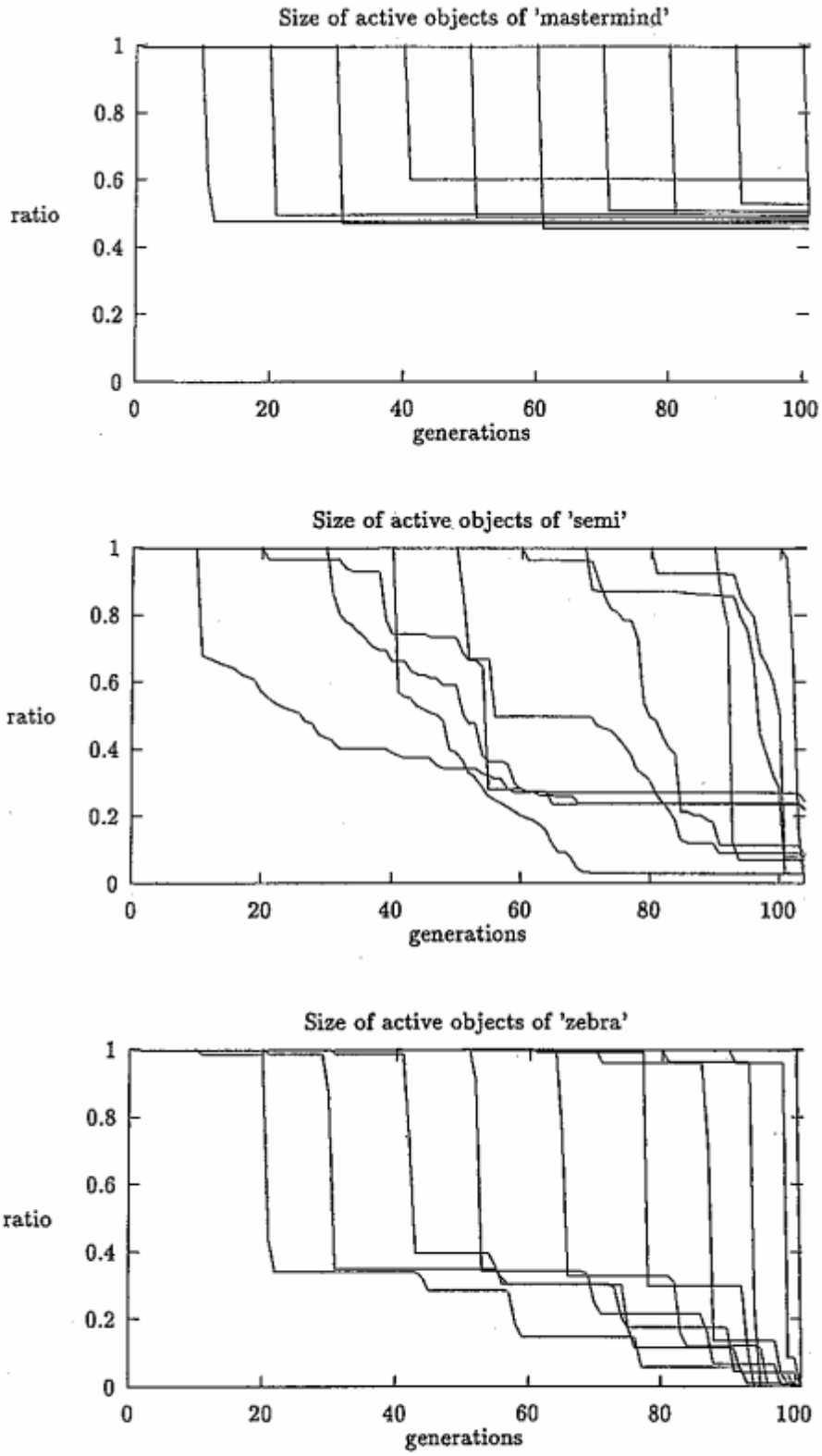

Figure 1: Lifetime Characteristics of KL1 Objects (continued)

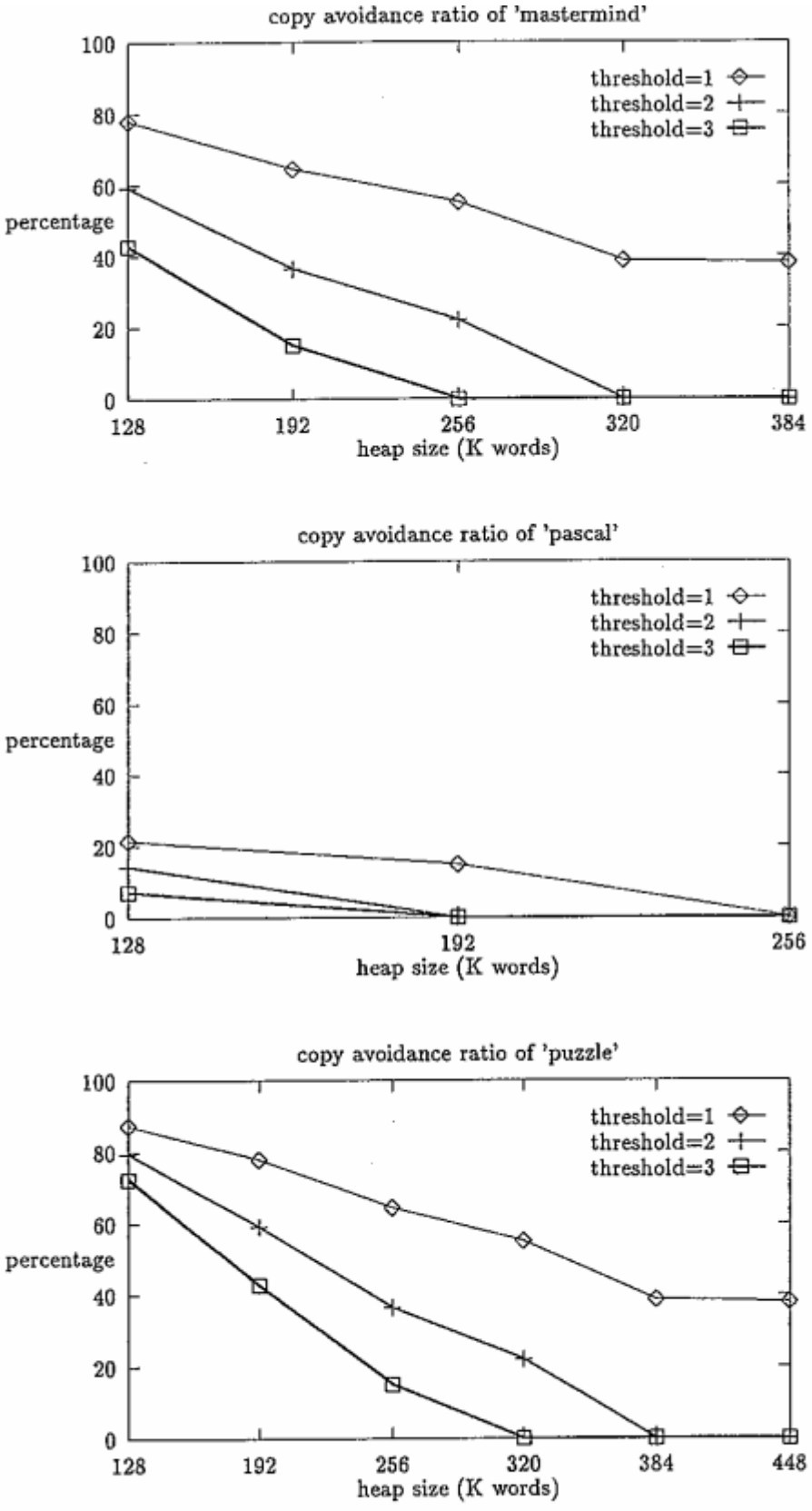

Figure 2: Savings Afforded by Two-Generation vs. Copying GC

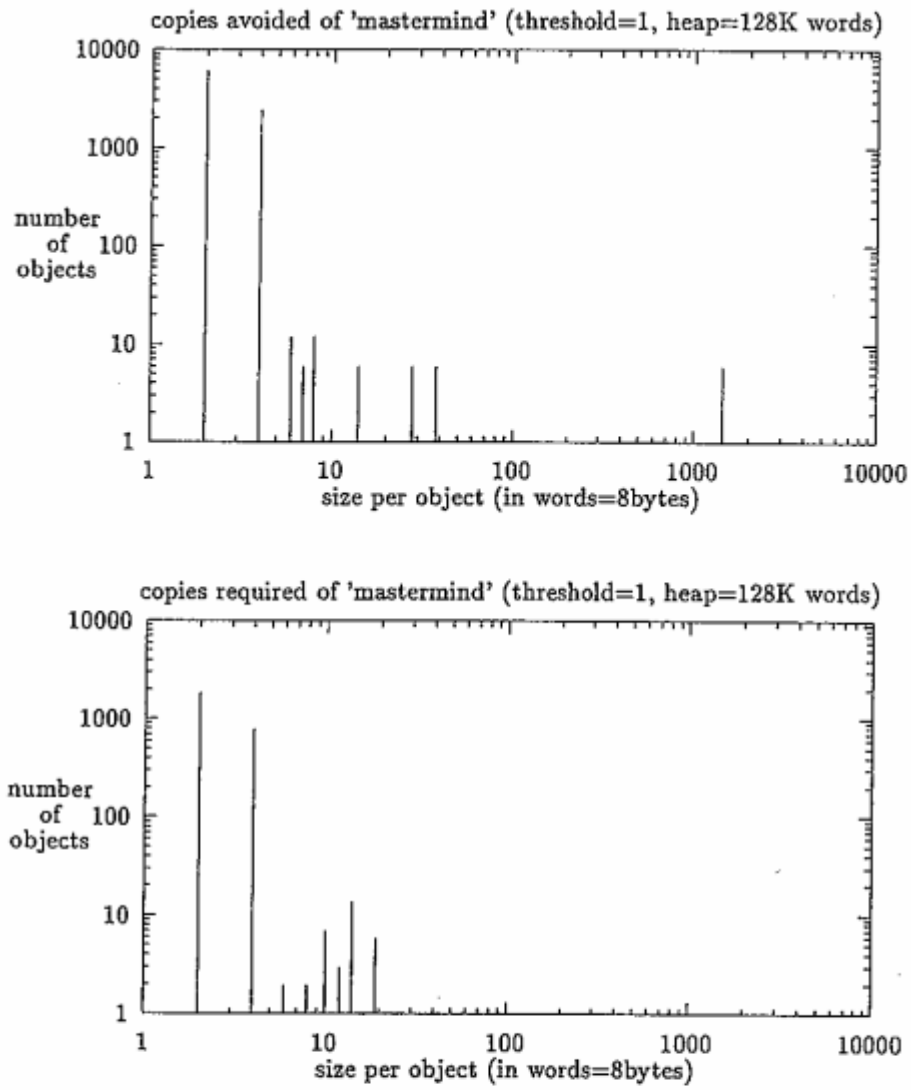

Figure 3: Distribution of Work per Object (Mastermind)

Curriculum Vita

Evan Tick
Assistant Professor
Computer and Information Science Department
University of Oregon
Eugene, Or 97403-1202.
As of May 1 1992

Education

1987	Ph.D. Electrical Engineering, Stanford University, Stanford CA.
1982	B.S. M.S. Electrical Engineering, Massachussets Institute of Technology, Cambridge MA.

Employment

1/90 present	Assistant Professor, Department of Computer and Information Science, University of Oregon.
10/88 9/89	Visiting Associate Professor, Research Center for Advanced Science and Technology (RCAST), University of Tokyo, Tokyo.
2/87 2/87	Visiting Researcher, ICOT, Tokyo
9/87 9/88	Research Associate, Computer Systems Laboratory, Stanford University, conducting research at Institute for New Generation Computer Technology (ICOT), Tokyo.
6/84 7/87	Consultant, Quintus Computer Systems, Inc., Palo Alto, CA.
9/82 6/87	Research and Teaching Assistant, Stanford University, Stanford CA.
6/83 9/83	Researcher, Center for Artificial Intelligence, Stanford Research Institute (SRI) International, Menlo Park CA.
6/81 6/82	Researcher, IBM T. J. Watson Research Center, Yorktown Heights, NY.
1/81 5/81	Teaching Assistant, Department of Electrical Engineering, Massachussets Institute of Technology, Cambridge MA.

Honors

Presidential Young Investigator's Award (NSF), 1990.

NSF ICOT Visitors Program, 1987 - 1988.

IBM Graduate Research Fellowship, 1985 - 1986.

Westinghouse National Science Talent Search, 4^{th} place, 1977.

Publications

Refereed Journals:

"Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor." (with A. Imai), *IEEE Transactions on Parallel and Distributed Computing*, in press.

"Memory and Buffer Referencing Characteristics of Prolog," *Journal of Logic Programming*, vol. 11, no. 2, 133-162, August 1991.

"Parallel Performance of Layered-Streams Programs," *International Journal of Parallel Programming*, Plenum Publishers, vol. 19, no. 6, 425-443, December 1990.

"Compile-Time Granularity Analysis of Parallel Logic Programming Languages," *Journal of New Generation Computing*, vol. 7, no. 2, 325-337, January 1990. (revision of paper in *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, November 1988).

"A Performance Comparison of Shared-Memory OR- and AND-Parallel Logic Programming Architectures for a Common Benchmark," *Journal of Information Processing*, vol. 13, no. 1, 62-71, Tokyo, 1990. (revision of paper in *Joint Symposium on Parallel Processing*, Atami, February 1989).

"Memory Referencing Characteristics and Caching Performance of AND-parallel Prolog on Shared-Memory Multiprocessors." (with M. V. Hermenegildo), *Journal of New Generation Computing*, vol. 7, no. 1, 37-58, 1989.

"Parallel Logic Programming in Prolog and FGHC," *IEEE Software*, vol. 6, no. 1, July 1989.

"Towards a Pipelined Prolog Processor," (with D. H. D. Warren), *Journal of New Generation Computing*, vol. 2, no. 4, 323-345, 1984. (revision of paper in *Proceedings of the International Symposium on Logic Programming*, Atlantic City, February 1984).

Books:

"Parallel Logic Programming," MIT Press, Cambridge MA, 1991.

"Memory Performance of Prolog Architectures," Kluwer Academic Publishers, Norwell MA., 1988. (revision of Ph.D. dissertation).