

Integrating Autotuning Into the Applications Development Environment

William Gropp

www.cs.uiuc.edu/homes/wgropp



What This Talk Is About

- Not about how to perform autotuning
- Not about what I have done in autotuning
- Rather, about what (I believe) needs to be done to make autotuning so easy for applications that it is used with more confidence that adding `-O` to the compile line
 - ◆ Much of this is usability
 - ◆ Some of it exposes additional interactions that need to be considered between autotuning tools



Where Would I Like to Use Autotuning and Code Transformations

- Now:
 - ◆ MPICH2
 - Datatype processing (loop optimizations, exploit alignment)
 - Currently includes some hand-optimized code
 - Common-case inlining
 - Structure layout optimization
 - ◆ PETSc
 - Multivector, matrix vector operations
 - Currently optimized for POWER (sort of)
 - Including data structure modifications
- Soon
 - ◆ Homogenous turbulence
 - ◆ QCD
 - ◆ Molecular dynamics
 - ◆ Additional libraries (pnetCDF, SPRNG, ...)
 - ◆ (See NSF Track 1 Call)



What Application Problem Are We Trying to Solve?

- Applications typically do not reach “achievable” performance
- Restated: Portable Programming for Performance
- What options are available for solving this problem?



Better Compiler

- ATLAS, PhiPAC, ... results show the futility of this
- Why: Many reasons, including
 - ◆ Need more information about runtime (loop limits, importance in big picture, data layout, ...)
 - ◆ Does not address data structure changes
 - Are ideas from aspect-oriented programming relevant?



Better Language

- All of the usual problems (ubiquity, adoption, support, permanence)
- Investment in current codes
- Current languages are not so bad, at least in terms of expressiveness
 - ◆ And have well-developed tool sets, expertise
- No one true language to rule them all



Libraries

- Limited to the functions provided by the library
- Composition of functions is hard
 - ◆ Performance penalty
 - Example: `BLAS;BLAS;BLAS;` has excessive memory motion
 - Example: Data-parallel-like routines, while simple, often imply extra synchronization, leading to scalability problems



The Preceding Suggests That

- Augmenting current programming models and languages to exploit advanced techniques for performance optimization (i.e., autotuning)
- Not a new idea, and some tools already do this.
- But how can these approaches become part of the mainstream development?



How Can Autotuning Tools Fit Into Application Development?

- In the short run, just need effective mechanisms to replace user code with tuned code
 - ◆ Manual extraction of code, specification of specific collections of code transformations
- But this produces at least two versions of the code (tuned (for a particular architecture and problem choice) and untuned). And there are other issues.
- What does an application want (what is the Dream)?



Application Needs Include

- Code must be portable
- Code must be persistent
- Code must permit (and encourage) experimentation
- Code must be maintainable
- Code must be correct
- Code must be faster

- I'll discuss each of these, followed by some implications.



Code Must be Portable

- Some version of code must run on all systems (ubiquity)
 - ◆ A stronger statement is “The fast code must be portable with performance”
- Must not require (but may exploit) special runtime systems
- Must not require special processing systems (or that system must be ubiquitous)



Code Must be Persistent

- The application should continue to run *well* even if the performance tools disappear or mutate
 - ◆ This can be considered a stronger version of the portability requirement
 - ◆ A stronger version again requires that the fast versions of the code persist even if the autotuning tools vanish
- This is necessary because of the poor record for persistence and backward compatibility of many tools
 - ◆ Yes, part of this is a result of insufficient funding, but some is self-inflicted through gratuitous changes in tool syntax and/or semantics



Code Must be Maintainable

- The developer must be able to edit the clean (original?) version of the code
- Changes to that code must be reflected in the tuned code (automagically)
 - ◆ Includes both simple changes (add a scalar term) and complex (change data structure to add fields, change order)
- Performance version(s) of code should be debugable
 - ◆ View both original code and “portable assembly” code



Code Must Permit Experimentation

- A stronger form of maintainability, requires robustness across significant changes in the code, including refactoring, changes in data structures, changes in algorithms
- It's the "original" form of the code that will be used for the experimentation



Code Must be Correct

- There must be a way to provide confidence in the performance transformations
- A combination of analysis and experiment are necessary, comparing against a “clean” version of the application
- Possible consequences of transformations (e.g., exploiting associativity) must be well documented and controllable on a per-instance basis
 - ◆ An example of a terrible, near evil idea - command line arguments that change the behavior of the language with file scope
 - Useful for benchmarks. Ghastly software engineering



Code Must be Faster

- The *whole* code, not just each piece
 - ◆ Beware, this has significant consequences
- And it must be relatively easy to document this
- And for all important platforms and datasets
- Note that applications are often large
 - ◆ No one autotuning tool may be adequate for the entire code
 - ◆ No one autotuning tool may be adequate for all input data, even for the same code
- Achieving this may require *collaboration* with the developer
 - ◆ Requesting additional information about general properties (e.g., index variable is a permutation, is monotone, is sparse, ...)
 - ◆ May require application-specific tuning



Implications of These Requirements

- Portable - augment existing language. Either use pragmas/comments or extremely portable precompiler
 - ◆ Best if the tool that performs all of these steps looks like just like the compiler, for integration with build process
- Persistent
 - ◆ Keep original and transformed code around
- Maintainable
 - ◆ Let use work with original code *and* ensure changes automatically update tuned code
- Correct
 - ◆ Do whatever the app developer needs to believe that the tuned code is correct
 - In the end, this will require running some comparison tests
- Faster
 - ◆ Must be able to interchange tuning tools - pick the best tool for *each* part of the code
 - ◆ No captive interfaces
 - ◆ Extensibility - a clean way to add new tools, transformations, properties, ...



Application-Relevant Abstractions

- Language for interfacing with autotuning must convey concepts that are meaningful to the application programmer
- Wrong: unroll
- Right (maybe): Performance precious, typical loop count between 100 and 10000, even, not power of 2



Training the Developer

- Autotuning tools are not magic
- Developers need feedback from autotuning tools about what
 - ◆ Inhibited performance
 - ◆ Increased search time
- Vector compilers trained users effectively
- Autotuning tools should be prepared to enter a dialog with user about better practices, opportunities



What Happens When the User's Code Is Changed?

- Correctness is the first requirement - may need to regenerate autotuned code
 - ◆ Must happen robustly and automatically
 - ◆ Analysis to determine if regeneration necessary (for correctness) or desirable (for performance)
 - ◆ Must fit smoothly in the user's build environment
 - Make if you are lucky, shell scripts or worse if not



Negotiations Between Tools

- Some performance enhancements require data structure modifications
 - ◆ What if the choice is not universally optimal?
 - The local optimum is not always the global optimum
 - ◆ How can autotuning components *negotiate* choices?
 - How is exponential explosion in choices avoided?
- Autotuning tools may need to be components in a larger system (which itself needs to be prepared to deal with evolving requirements, tools, abstractions)



Possible Requirements (1)

- User markup of code with appropriate abstractions
 - ◆ Properties of code to be tuned, not transformations to be applied
 - ◆ Original code always runs
 - ◆ Independent of any particular tool
 - To allow multiple providers
- Automatic extraction *and consistency management* of code to be tuned
 - ◆ Includes automatic verification (possibly in a weak sense) of equivalence between original and tuned code
 - ◆ Developer can work with original code or with tuned code
 - Changes must be kept consistent



Possible Requirements (2)

- Integration with language
 - ◆ Exploit type information; safely manage extensive data structure transformations
 - Correctness - need to detect failure by user to correctly mark up code
 - Provide option for runtime checks of asserted properties
- Extensibility
 - ◆ Avoid *brittleness* in tuning
 - Application- or domain-specific extensions, written by experts, may be needed
 - Tool independence - Let developer combine tools and let them compete for the best tuned version (different tools may excell on different platforms)



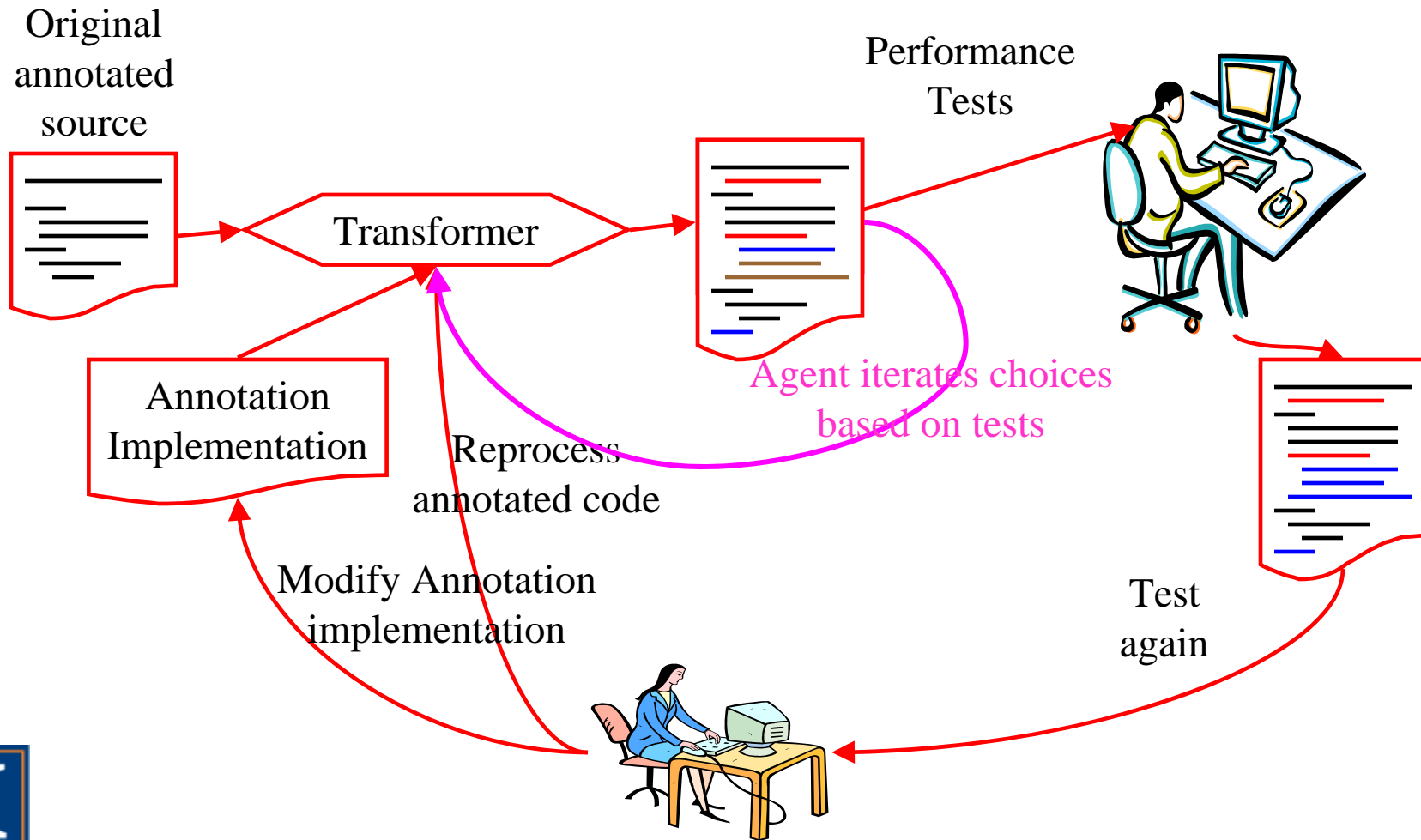
What Might an Interface Look Like?

- Structured comments around “performance precious” code
 - ◆ Notations about problem size, monotonicity, *not* unroll amounts
 - ◆ System must find relevant declarations (requires compiler infrastructure)
 - ◆ Hash or other technique used to detect changes (invalidate tuned code)
 - ◆ Other abstractions help with defining relevant tests
- Option to retain original and tuned code in a replacement file
 - ◆ Provides a measure of portability
- Tool independence
 - ◆ Should be able to let tools compete, on a block-by-block basis, for best tuned code
 - ◆ Easy to confirm that tuned code is correct
 - Must address floating point issues - bitwise identity is not always a requirement
- Generated code contains sufficient hooks for debugger to relate back to original code
- “Database” of tuned versions
 - ◆ Possibly in an auxiliary file, to make it easy to move project
 - Even better if we could get away from files as containers, but that’s another story



Code Development Cycle

- Permit *evolution* of the transformed code



Conclusions

- Adoption of autotuning requires at least two types of advancements
 - ◆ Usability issues to make use of an individual autotuning tool as easy as changing the compiler optimization
 - ◆ Solving the problem of integrating (including *composing*) different autotuning components into the applications build system, including negotiations between tools
- Other developments have similar needs
 - ◆ E.g., Heterogeneous hardware (GPGPU, FPGA, etc.) programming
 - ◆ We need a common system for managing language and tool components (not just autotuners)

